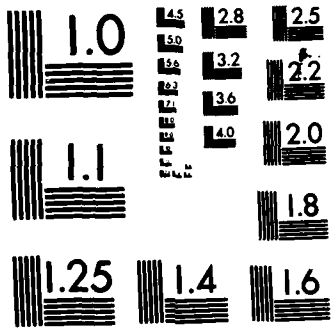
AD-A159 189 RESEARCH TO STUDY SPECIFIC IMPORTANT PROBLEMS IN DISTRIBUTED SYSTEMS AND (U) TEXAS UNIV AT AUSTIN DEPT OF COMPUTER SCIENCES K M CHANDY ET AL. JUL 85 1/2

UNCLASSIFIED AFOSR-TR-85-0732 AFOSR-81-0205 F/G 9/2 NL



MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS-1963-A

AFOSR-TR- 85-0732

(2)

AD-A159 189

Air Force Scientific Report

AFOSR 81-0205

6/15/81 through 6/14/85

K. M. Chandy and J. Misra

DTIC FILE COPY

DEPARTMENT OF COMPUTER SCIENCES
THE UNIVERSITY OF TEXAS AT AUSTIN
AUSTIN, TEXAS 78712

85 9 10 081

A159 184

Air Force Scientific Report

AFOSR 81-0205

6/15/81 through 6/14/85

K. M. Chandy and J. Misra

Department of Computer Sciences
The University of Texas
Austin, Texas 78712
(512) 471-4353

19 July 1985

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER AFOSR-TR- [illegible] 0732	2. GOVT ACCESSION NO. AD-A159189	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) Final Scientific Report for AFOSR grant AFOSR 81-0205		5. TYPE OF REPORT & PERIOD COVERED final - 6/14/81 - 6/15/85
		6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s) Professor K. M. Chandy Professor J. Misra University of Texas at Austin		8. CONTRACT OR GRANT NUMBER(s) AFOSR 81-0205
9. PERFORMING ORGANIZATION NAME AND ADDRESS Computer Sciences Department University of Texas at Austin Austin, Texas 78712		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS 61102F 2304 A2
11. CONTROLLING OFFICE NAME AND ADDRESS AFOSR/NM Bolling AFB, DC 20332		12. REPORT DATE July, 1985
		13. NUMBER OF PAGES 170
14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office) AFOSR/NM Bldg 410 Bolling AFB DC 20332-6448		15. SECURITY CLASS. (of this report) UNCLA[illegible]
		15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release;
distribution unlimited.

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

Our work has resulted in a number of significant algorithms for distributed systems. Notable among these are, (1) Distributed Snapshots: which allows for the construction of a consistent global state, (2) The Drinking Philosophers Problem: which captures the essence of many conflict resolution problems, (3) Detection of Quiescent Properties: which allows detection of many "stable properties" without taking a snapshot and (4) Distributed Search: which allows for the solution of dynamic programming problems on a message passing architecture.

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE

A159189

Research Objectives

The goal of our research during 1981-83 was to study specific, important problems in distributed systems and propose solutions for them. Intuitive arguments about distributed algorithms are error-prone and hence we started work on proving the correctness of distributed systems. The success of our research effort during this period led us to set some ambitious goals for 1983-85: to identify unifying paradigms in distributed computing regardless of the underlying architecture, communication mechanism or operating system. We have concentrated on problems that are general (applicable to a variety of systems), and fundamental (their solutions constitute the critical portions of fugure and existing systems). We have not addressed problems for specific operating systems or architectures. Our objective is to make our research applicable for novel architectures that we may see in the next decade.

The development of distributed programs is much more difficult than the development of sequential programs. One of the difficulties in distributed programming is the absence of a set of general paradigms. Sequential programming has its paradigms such as backtracking, divide and conquer and dynamic programming. Distributed programming paradigms are often confused with specific problems of the underlying architecture or operating system. One of the first genuine paradigms for distributed systems is Lamport's event ordering algorithm. This has found application in distributed mail systems, distributed databases and operating systems. Unfortunately, very few such problems have been identified to be of general importance in distributed systems area and fewer still have been solved (correctly). Our study revealed that almost *all* deadlock detection algorithms published prior to ours (in 1982) either failed to report deadlock where one existed or reported deadlock where none existed.

We proposed to study asynchronous message passing systems, consisting of arbitrary number of processors (hence to be called, processes), arbitrary interconnections among processes and arbitrary, finite delays for message communication. The paradigm of *learning* is fundamental to distributed systems: algorithms are executed so that processes "learn" properties about the underlying system. What is learned may be static - an unchanging property - or dynamic - a property changing with time. A class of *static* problems arise from the network structure: (1) each process learns the topology of the network, (2) assign unique id's to each process, and (3) a process learns the shortest path between two processes, etc. We have, however concentrated on *dynamic* problems; arbitrary computations are assumed to be proceeding at various processes and it is required to superimpose an algorithm on the underlying computations so that a process learns a dynamic property of the system such as: is the system deadlocked? The dynamic problem is considerably more difficult than the static problem.

Our work pointed out the need for formal methods in reasoning about communicating processes. Verification techniques for sequential systems are sometimes unnecessary if proper care is taken in structuring and developing the system. Unfortunately, nondeterministic control makes it nearly impossible to apply similar reasoning based on locus of control; entirely new techniques were called for. One of our goals was to propose techniques which not only are useful, but are also usable: we planned to prove complex systems using these techniques.

We elaborate our major contributions in the following pages.

Status of Research

Our research has resulted in a number of significant contributions which have already appeared, or will appear, in literature. Notable among our achievements are the following:

1. **Reasoning Techniques for Distributed Systems:** We were the first ones to propose "compositional" proof techniques for distributed systems, which allowed the proof a distributed system to be partitioned into proofs of individual components making up the system. This structuring of proof made it possible to prove complex systems. Current work in parallel program verification by most researchers has this compositional flavor.

2. **Distributed Snapshot:** Many problems in distributed systems require that a "snapshot" of the system be taken. We want to record the states of all channels and processes at some instant. The problem is that in distributed systems there is no way of synchronizing such an "instant." If we could take global snapshots, all dynamic problems reduce to static problems because we can take a sequence of snapshots of the dynamic system and analyze each snapshot in turn. Since each snapshot is static the analysis of a snapshot is a static problem. The solution of this problem subsumed a large body of work on termination/deadlock detection and distributed checkpointing.

3. **Detection of Quiescent Properties:** Detection of certain properties, such as termination or absence of tokens, can be accomplished more efficiently than by taking distributed snapshots, as in (1). We give a characterization of a class of properties, called *quiescent properties*, and show how their presence in a system can be detected.

4. **The Drinking Philosophers Problem:** This captures the essence of conflicts - two or more processes are prevented from continuing their executions in order to satisfy certain system constraints - and their resolution in many distributed programming situations. Our notion of conflict

is general enough to include such legendary problems as mutual exclusion, dining philosophers and multiple copy updates, as special problems. Our solution shows the importance of introducing and preserving an asymmetry among processes.

5. **Distributed Search:** This provides a general strategy for implementations of dynamic programming solutions in a distributed system.

We elaborate each of these contributions and our other work under this grant, in the following pages.

Reasoning Techniques: (Our later work in this area has been supported completely by AFOSR)

If a number of processes execute concurrently, it is difficult to make statements about the ensemble, because the program control resides simultaneously at many different points, one point in each process. It is even more difficult to prove properties of a process in isolation from its environment. However, this is exactly what is required if we ever hope to substitute one process by another without affecting the functioning of the whole system.

We pioneered the area of *compositional proof systems* in which each process has a specification independent of its environment. We showed how the specifications of component processes can be combined to yield a specification for the system, as a whole. This made it possible to structure the proof of a system along the lines in which a system is structured into processes. Previous proof techniques required elaborate "noninterference proofs", to show that functioning of one component would not be affected by simultaneous functioning of another component; this required not only the specification of components but also their inner structure. Consequently such proofs tended to be long. Our proof technique has been applied by Ossefort ("Correctness Proofs of Communicating Processes - Three Illustrative Examples from the Literature," *ACM TOPLAS*, Vol. 5, No. 4, October 1983, pp. 620-640). in proving several complex distributed algorithms. This work has been extended in [5] and [19].

Distributed Snapshots

A problem of considerable importance in distributed data bases, where a process may have *locked* some data items and is waiting for others, is the problem of *deadlock*. A similar problem appears in distributed routing (deadlock due to insufficient buffers) or legitimately, in a distributed computation where the processes have run out of data. A somewhat different problem is to take a checkpoint of a distributed system; such checkpoints are necessary for rollback and recovery in machines like the Cosmic Cube (built by Professor Chuck Seitz at Caltech).

Each of these problems can be solved by taking a *snapshot* of the system. A snapshot is a state of the system: states of processes and channels linking the processes, which could have arisen at some point in the past.

An algorithm for taking a distributed snapshot was developed by one of the principal investigators (Chandy) and Leslie Lamport. This algorithm requires minimal overhead, does not interfere with the underlying computation and is easy to implement. This work has been widely referenced and has been developed further by E. W. Dijkstra.

The Drinking Philosophers Problem

Conflicts arise in distributed systems due to contentions for shared resources. For instance, two processes cannot write into a shared data item simultaneously, two machines cannot broadcast messages on an ethernet at the same time, etc. Conflicts, such as these, are typically resolved either (1) by a central process or (2) by assigning static, global priorities to processes or (3) by resolving to probabilistic decision making by individual processes. We identified the basic ingredient of every non-probabilistic solution: *asymmetry among processes.* We showed how asymmetry can be introduced initially by judicious of shared resources and how to preserve asymmetry in a fair manner. Our formulation of the problem starts with (1) a set of processes, (b) a set of resources shared among some subset of processes and (3) an arbitrary computation at the processes which result in requests for the resources. The mutual exclusion problem is a special case where there is a single resource shared by all processes. The Dining Philosophers Problem has resources (forks) shared by exactly two processes and requests for resources are always for identical sets of resources. Our solution initially assigned resources to processes in such a manner that individual processes could be distinguished by the resources that they possess. We proposed certain rules for relinquishing resources which preserved this asymmetry. Our solution is very efficient, because processes make only local transformations and they send no messages unless they are requesting or relinquishing resources. In fact, our solution to the dining philosophers problem can be shown to be optimal in the number of messages.

Detection of Quiescent Properties

Quiescence properties of a distributed system are those which continue to be true once they become true. Termination, deadlock, and absence of tokens in a system are examples of such properties. These properties may be detected by applying the distributed snapshot algorithm described earlier. However, we show that there is a more efficient class of algorithms, which includes a number of published algorithms as special cases, for these classes or problems.

Our algorithm is given in fairly abstract terms using certain unspecified conditions. Different instances of these conditions result in different algorithms. Our development of the algorithm, using stepwise refinement, led to very real conditions under which the algorithm can operate.

Distributed Search

A large class of optimization problems have the following structure: a problem may have many feasible solutions and, of these, we seek the solution with the lowest *cost*. The critical task, therefore, is to find a feasible solution whose cost is bounded below some given threshold; as the threshold is lowered, an optimum solution is approximated.

This problem, again, is of a very general nature. It includes such well known problems as the traveling salesman problem and shortest path problems. The generalization consists of making very few assumptions about the problem structure. This work is similar in spirit to a very general model of dynamic programming introduced by Karp and Held (Siam Journal of Applied Mathematics, May 1967). It is hoped that the proposed paradigm will include all deterministic, search based optimization procedures.

Other Related Work

Our work on distributed simulation, partially funded by AFOSR, is considered to be the seminal work in that area. We pioneered the area by demonstrating that system simulations can run efficiently on several parallel machines. A large number of researchers in U.S.A., Europe and Japan are now working in this area. It is not too much to expect that *all* large scale simulations have to be distributed in the future. Our problem formulation and solution procedures were general - independent of specific properties of the system being simulated or idiosyncrasies of the underlying architecture - which allow it to be adapted for specific problems and architectures.

List of Publications

[1] "Distributed Computation on Graphs: Shortest Path Algorithms," *Communications of the ACM*, Vol. 25, No. 11, November 1982, pp. 833-837, (K. Mani Chandy and Jayadev Misra)

[2] "Distributed Deadlock Detection," *ACM Transactions on Computer Systems*, Vol. 1, No. 2, May 1983, pp. 144-156, (K. M. Chandy, J. Misra and L. Haas).

[3] "A Distributed Graph Algorithm: Knot Detection," *ACM Transactions on Programming Languages and Systems*, Vol. 4, No. 4, October 1982, pp. 678-686, (J. Misra and K. M. Chandy).

[4] "A Distributed Algorithm for Detecting Resource Deadlocks in Distributed Systems," *Proceedings of the ACM SIGACT-SIGOPS Conference on the Principles of Distributed Computing*, August 18-20, 1982, Ottawa, Canada, (K. M. Chandy and J. Misra).

[5] "Proving Safety and Liveness of Communicating Processes with Examples", *Proceedings of the ACM SIGACT-SIGOPS Conference on the Principles of Distributed Computing*, August 18-20, 1982, Ottawa, Canada, (J. Misra, K. M. Chandy and T. Smith).

[6] "Finding Repeated Elements," *Science of Computer Programming*, No. 2, (1982), pp. 143-152, North-Holland Publishing Company, (J. Misra and D. Gries).

[7] "Assigning Processes to Processors in Distributed Systems," *Proceedings of the 1983 International Conference on Parallel Processing*, August 23-26, 1983, Bellaire, Michigan, (Elizabeth Williams).

[8] "Paradigms for Distributed Computing," Invited paper Third International Conference on Foundations of Software Technology and Theoretical Computer Science, Bangalore, India, December 12-14, 1983, (K. M. Chandy).

[9] "Distributed Simulation," Tutorial presented at the IEEE Computer Society 4th International Conference on Distributed Computing Systems, May 14-18, 1984, San Francisco, California (Jayadev Misra).

[10] "Processor Queueing Disciplines in Distributed Systems," *Proceedings of the 1984 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, August 21-24, 1984, Cambridge, Massachusetts, (Elizabeth Williams).

[11] "The Effect of Queueing Disciplines on Response Times in Distributed Systems," *Proceedings of the 1984 International Conference on Parallel Processing*, August 22-24, 1984, Bellaire, Michigan, (Elizabeth Williams).

[12] "The Drinking Philosophers Problem," *ACM Transactions on Programming Languages and Systems*, Vol. 6, No. 4, October 1984, pp. 632-646, (K. M. Chandy and J. Misra).

[13] "Distributed Snapshots: Determining Global States of Distributed Systems," *ACM Transactions on Computer Systems*, Vol. 3, No. 1, February 1985, pp. 63-75, (K. M. Chandy and Leslie Lamport).

Papers written since Annual Report for year 1983-84

[14] "On Distributed Search", to appear in Information Processing Letters, (Ted Herman and K. Mani Chandy).

[15] "A Paradigm for Detecting Quiescent Properties in Distributed Computations," NATO ASI Series, F13, Springer-Verlag Lecture Notes in Computer Science, to appear in 1985, (K. Mani Chandy and Jayadev Misra).

[16] "An Example of Stepwise Refinement of Distributed Programs: Quiescence Detection," to appear in *ACM Transactions on Programming Languages and Systems*, (K. Mani Chandy and Jayadev Misra).

[17] "A Class of Termination Detection Algorithms for Distributed Computations," Technical Report TR-85-07, The University of Texas at Austin, Computer Sciences Department, May 1985, (Devendra Kumar), submitted to IEEE Transactions on Software.

[18] "A Model and Proof System for Asynchronous Networks," *Proceedings of the 4th ACM SIGACT-SIGOPS Conference on the Principles of Distributed Computing*, Minaki, Canada, August 5-7, 1985, (Bengt Jonsson).

[19] "A Novel Approach to Sequential Simulation," Technical Report TR-85-14, The University of Texas at Austin, Computer Sciences Department, July 1985, (Devendra Kumar), submitted to IEEE Transactions on Software.

[20] "A High Speed Distributed Simulation Scheme and Its Performance Evaluation," (Devendra Kumar), submitted to the Nineteenth Annual Simulation Symposium and CMG '85.

List of Professional Personnel

Name	K. Mani Chandy
Title	Co-Principal Investigator
Department	Faculty, Department of Computer Sciences, UT
Name	Jayadev Misra
Title	Co-Principal Investigator
Department	Faculty, Department of Computer Sciences, UT
Name	Marty Ossefort {Note: received 3 months of support}
Title	Graduate Student {Graduated August, 1982}
Department	Graduate School, Department of Computer Sciences, UT
Name	Devendra Kumar
Title	Graduate Student
Department	Graduate School, Department of Computer Sciences, UT
Name	Elizabeth Williams
Title	Graduate Student {Graduated May, 1983}
Department	Graduate School, Department of Computer Sciences, UT
Name	Bob Comer
Title	Graduate Student
Department	Graduate School, Department of Computer Sciences, UT
Name	Ted Herman
Title	Graduate Student
Department	Graduate School, Department of Computer Sciences, UT

Degrees Awarded

Recipient	Elizabeth Williams
Award Date	May, 1983
Type of Degree	Ph.D (Doctor of Philosophy)
Thesis Title	"Design, Analysis, and Implementation of Distributed Systems from a Performance Perspective"
Department	Department of Computer Sciences

Recipient	Martin John Ossefort (3 months of support)
Award Date	August, 1982
Type of Degree	Ph.D (Doctor of Philosophy)
Thesis Title	"A Unified Approach to Formal Verification of Network Safety Properties"
Department	Department of Computer Sciences

Degrees Expected

Recipient	Ted Herman
Award Date	December, 1985
Type of Degree	Ph.D (Doctor of Philosophy)
Thesis Title	"Paradigms for Distributed and Parallel Programming"
Department	Department of Computer Sciences

Interactions (Invited lectures listed for the period 1983-85 only)

Invited lectures presented by Professor K. Mani Chandy on topics related to work performed under this grant.

• Cornell University	May 2-6, 1983
• Stanford University	November 7, 1983
• M.I.T.	November 13, 1983
• Third Conference on Foundations of Software Technology and Theoretical Computer Sciences, Bangalore, India	December 12-14, 1983
• University of California at Berkeley	February 23, 1984
• Distinguished Lecture Series, University of Minnesota	May 14-15, 1984
• IBM Research Lab, Yorktown Heights	July 11-13, 1984
• Keynote address, 3rd ACM Principles of Distributed Computing Conference, Vancouver, Canada	August 26, 1984
• Pennsylvania State University	October 1-5, 1984
• Distinguished Lecture Series, University of Central Florida, Orlando	March 9-17, 1985
• DEC, Systems Research Center, Palo Alto	March 21, 1985
• IFIP W.G. 2.3, Manchester, England	April 13-18, 1985

Invited lectures given by Professor J. Misra on topics related to work performed under this grant.

• University of California at Berkeley	October 18, 1983
• University of Manchester, England	November 9, 1983
• University of California at Los Angeles	January 26, 1984
• IBM Research Labs, Yorktown Heights	February 9, 1984
• IBM Research Labs, San Jose	March 7, 1984
• California Institute of Technology	March 27, 1984
• IEEE Fourth International Conference on Distributed Computing Systems, Invited Tutorial entitled, "Distributed Simulation"	May 14-18, 1984
• Xerox Palo Alto Research Center Palo Alto, California	March 15, 1984
• University of Washington	May 10, 1984
• U.S. - U.K. Joint Workshop on Concurrency, Carnegie-Mellon University	July 9-11, 1984
• IFIP W.G. 2.3, Victoria, Canada	July 23-27, 1984
• Workshop on Reasoning About Cooperating Agents and Concurrent Processes, SRI, Monterey Dunes, California	August 22-24, 1984
• Yale University	September 27, 1984
• Advanced NATO Study Institute on Logics and Models for Verification and Specification of Concurrent Systems, France	October 8-19, 1984
• Carnegie-Mellon University	April 11, 1985

• Cornell University	April 12, 1985
• IFIP W.G. 2.3, Manchester, England	April 15-19, 1985

Advisory Functions

Dr. Chandy serves on the Committee on Recommendations for U.S. Army Basic Scientific Research, National Research Council, July 1, 1984 to June 30, 1987.

Additional Statements

One measure of importance of a piece of research is its acceptance by the scientific community at large. Typically, a successful piece of work gains wide acceptance within three to five years of its publication. Our work on Distributed Simulation, funded partly by the Air Force is now considered to be one of the most important developments in that area (as evidenced by the number of workers and publications). Similarly, our work on Reasoning About Communicating Processes (funded by AFOSR) remains one of the most referenced papers in that area; in a recent book - A Survey of Verification Techniques for Parallel Programs by Howard Barringer (Lecture Notes in Computer Science, 191, Springer-Verlag, 1985) - comparing nine different methods for parallel program verification, our approach is listed as one of the four that admits of hierarchical developments.

We expect the other work reported here - particularly, Distributed Snapshots and Drinking Philosophers Problem - to have the same kind of impact in the next three years or so. We have been invited by at least thirty universities and research labs to deliver lectures related to these topics, in the last two years. We infer that our work has gained wide acceptance because we have emphasized issues that are fundamental and not only of immediate practical interest.

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER	2. GOVT ACCESSION NO.	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) Manuscript: "On Distributed Search"		5. TYPE OF REPORT & PERIOD COVERED final: 6/14/81 - 6/15/85
		6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s) Professor K. M. Chandy Ted Herman (Graduate Student) University of Texas at Austin		8. CONTRACT OR GRANT NUMBER(s) AFOSR 81-0205
9. PERFORMING ORGANIZATION NAME AND ADDRESS Computer Sciences Department University of Texas at Austin Austin, Texas 78712		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
11. CONTROLLING OFFICE NAME AND ADDRESS Capt A. L. Bellamy AFOSR/NM Bolling AFB, DC 20332		12. REPORT DATE July 1985
		13. NUMBER OF PAGES
14. MONITORING AGENCY NAME & ADDRESS(if different from Controlling Office)		15. SECURITY CLASS. (of this report)
		15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

to appear in Information Processing Letters

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

Distributed computation, graph algorithms

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

Many optimization problems have the following structure: A problem may admit many feasible solutions, and of these, one seeks the solution of lowest possible cost. In this paper, we consider the task of finding a feasible solution whose cost is bounded below some given threshold. As the threshold is lowered, an optimum solution is approximated.

Our distributed program is a parallel search for the approximate solution. The underlying distributed system is an asynchronous network of processes, with

indeterminant timings between computations. The distributed search therefore has this non-deterministic flavor: The computation advances whenever and wherever possible in the distributed system.

On Distributed Search

Ted Herman and K. Mani Chandy
Department of Computer Science
University of Texas at Austin

Keywords: Distributed computation, graph algorithms

* This research was supported by a grant from the Air Force Office of Scientific Research under grant AFOSR 81-0205.

1.0 *Introduction*

Many optimization problems have the following structure: A problem may admit many feasible solutions, and of these, one seeks the solution of lowest possible cost. In this paper, we consider the task of finding a feasible solution whose cost is bounded below some given threshold. As the threshold is lowered, an optimum solution is approximated. This work is based, in part, on the pioneering work of Karp and Held [3] on dynamic programming.

Our distributed program is a parallel search for the approximate solution. The underlying distributed system is an asynchronous network of processes, with indeterminant timings between computations. The distributed search therefore has this non-deterministic flavor: The computation advances whenever and wherever possible in the distributed system.

2.0 *Thr Distributed System*

The distributed system consists of a fixed set of processes that communicate solely by passing messages. A process may directly send a message to another process only if there exists a channel between the two. A message may contain arbitrary data. The channels are loss-less: If process B sends a message to process D, then we can assert that process D will receive the message (perhaps after some finite delay). The number and locations of channels is fixed in the distributed system. It is useful to think of processes as vertices and to channels as edges of a finite graph. The set of processes in the system is

$V = \{v_i\}, i = 1,...,n$

and (v_i,v_j) denotes a channel/edge. The set E is the set of channels defined for the distributed system. If there is a channel between a pair of processes, then we call these processes *neighbors*.

3.0 *The Approximation Problem*

Let P be the problem to be approximated. The solution to P will be an item called a *policy*. Policies that satisfy the basic contraints of P are *feasible* policies. Define POLICY as the set of all possible policies. Then for x∈POLICY, the predicate function FEAS determines feasibility: FEAS(x) is true iff x is a feasible policy. Messages between processes will represent policies. Processes receive policies, amend them, and transmit them during the computation.

Policies are ordered by a cost function. COST(x) yields a number that is the cost of policy x. Policy x is *acceptable* if it satisfies

FEAS(x) and $\mathrm{COST}(x) < R$,

where R is a number defining the level of approximation desired for problem P. Further restrictions on the COST function are revealed in Section 4.

There is a bijection from the set of processes V to a set of functions Z. Elements of Z are functions that map policies to policies. For convenience, let z_i correspond to process v_i.

The class of approximation problems suited to our analysis must have solutions expressible as compositions of functions in Z. Let $W(v_j)$ represent some finite walk, originating at v_j, over edges of the graph defined by the distributed system. $W(v_j)$ can be written in the form

$v_j v_{i(1)} v_{i(2)} \dots v_{i(k-1)} v_{i(k)}$

where the sequence i(1), i(2), ..., i(k) designates the order of vertices in the walk. Corresponding to $W(v_j)$, we write the composition

$S(W(v_j),x) = z_{i(k)} \circ \dots \circ z_{i(2)} \circ z_{i(1)}(x)$,

which is a generation sequence based on x generated by v_j. The *empty* walk contains no edges. $S(W(v_j),x) = x$ for the empty walk.

3.1 *Solution Characterization*: A solution to P is an item y satisfying:
(a) FEAS(y) and COST(y)<R, or
(b) FEAS(y) and
{For all $w \in$POLICY, FEAS(w) implies COST(y)$\leq$COST(w)}, or
(c) $y=\perp$ and {For all $w \in$POLICY, FEAS(w) is *false*}.

The first case provides an acceptable policy. In the second case no acceptable policy can be found, so a minimum cost policy is the result. In the last case, the output is a special symbol $\perp$, which indicates that no feasible solution to P is possible. By convention COST($\perp$) = ∞. When a process v_i sends a message y to some other process, then v_i is said to *generate* y. We say a distributed computation generates y if any v_i in the system generates y during the computation. The class of approximation problems to be considered can now be precisely characterized.

3.2 *Proposition*: A solution to approximation problem P, P = [V,E,Z,FEAS,COST,R], can be generated by a distributed computation, provided that a solution to P is expressible as $S(W(v_*),\delta)$, for some selection of $W(v_*)$, where v_* is the label of a vertex called the *initiator*, and δ is some initial policy.

4.0 *Distributed Algorithm*

We expose the distributed algorithm in successive refinements. First, a simple procedure will suffice to generate an acceptable policy. Subsequent procedures generate solutions with greater economy. Then termination criteria are introduced to complete the algorithm.

The computation will begin at the initiator v_*. The initiator will originate the distributed computation by sending policy δ to its neighbors, following proposition (3.2). Let MESSAGE(i,j,x) denote the event: v_j receives policy x, sent by v_i. The behavior of v_j following this event is described by the following procedure.

4.1 *Procedure*: Upon MESSAGE(i,j,x) v_j sends $z_j(x)$ to all neighbors.

4.2 *Lemma*: The procedure 4.1 generates a solution to problem P.
Proof. By definition, the specification to problem P includes a set of functions Z such that a solution to P is in the form $S(W(v_*),\delta)$. Since procedure 4.1 generates all possible $W(v_*)$, a solution to P must be generated. ■

where R is a number defining the level of approximation desired for problem P. Further restrictions on the COST function are revealed in Section 4.

There is a bijection from the set of processes V to a set of functions Z. Elements of Z are functions that map policies to policies. For convenience, let z_i correspond to process v_i.

The class of approximation problems suited to our analysis must have solutions expressible as compositions of functions in Z. Let $W(v_j)$ represent some finite walk, originating at v_j, over edges of the graph defined by the distributed system. $W(v_j)$ can be written in the form

$v_j v_{i(1)} v_{i(2)} \dots v_{i(k-1)} v_{i(k)}$

where the sequence i(1), i(2), ..., i(k) designates the order of vertices in the walk. Corresponding to $W(v_j)$, we write the composition

$S(W(v_j),x) = z_{i(k)} \circ \dots \circ z_{i(2)} \circ z_{i(1)}(x)$,

which is a generation sequence based on x generated by v_j. The *empty* walk contains no edges. $S(W(v_j),x) = x$ for the empty walk.

3.1 *Solution Characterization*: A solution to P is an item y satisfying:

(a) FEAS(y) and $COST(y) < R$, or
(b) FEAS(y) and
 {For all $w \in$ POLICY, FEAS(w) implies $COST(y) \leq COST(w)$}, or
(c) $y = \perp$ and {For all $w \in$ POLICY, FEAS(w) is *false*}.

The first case provides an acceptable policy. In the second case no acceptable policy can be found, so a minimum cost policy is the result. In the last case, the output is a special symbol $\perp$, which indicates that no feasible solution to P is possible. By convention $COST(\perp) = \infty$. When a process v_i sends a message y to some other process, then v_i is said to *generate* y. We say a distributed computation generates y if any v_i in the system generates y during the computation. The class of approximation problems to be considered can now be precisely characterized.

3.2 *Proposition*: A solution to approximation problem P, P = [V,E,Z,FEAS,COST,R], can be generated by a distributed computation, provided that a solution to P is expressible as $S(W(v_*),\delta)$, for some selection of $W(v_*)$, where v_* is the label of a vertex called the *initiator*, and δ is some initial policy.

4.0 *Distributed Algorithm*

We expose the distributed algorithm in successive refinements. First, a simple procedure will suffice to generate an acceptable policy. Subsequent procedures generate solutions with greater economy. Then termination criteria are introduced to complete the algorithm.

The computation will begin at the initiator v_*. The initiator will originate the distributed computation by sending policy δ to its neighbors, following proposition (3.2). Let MESSAGE(i,j,x) denote the event: v_j receives policy x, sent by v_i. The behavior of v_j following this event is described by the following procedure.

4.1 *Procedure*: Upon MESSAGE(i,j,x) v_j sends $z_j(x)$ to all neighbors.

4.2 *Lemma*: The procedure 4.1 generates a solution to problem P.
Proof. By definition, the specification to problem P includes a set of functions Z such that a solution to P is in the form $S(W(v_*),\delta)$. Since procedure 4.1 generates all possible $W(v_*)$, a solution to P must be generated. ■

Note that procedure 4 1 does not terminate, nor does it recognize a solution to P. Lemma 4.2 only states that some process v_i in the distributed system will, at some point, send a message $z_i(x)$, where $z_i(x)$ is a solution to P.

It is important to consider the effect that an individual message has on the course of the distributed computation. For example, if a message x can be removed from a computation and P is solved anyway, then message x should not be generated for reasons of efficiency. The next results develop apparatus needed to decide when a policy x should be discarded.

4.3 *Procedure*: Upon MESSAGE(i,j,x), if there is no solution to P of the form $S(W(v_j),z_j(x))$, for any $W(v_j)$, then v_j sends no messages. Otherwise procedure 4.1 is invoked.

Temporarily we focus on P where $R=\infty$. P is therefore a search for any feasible policy. Let $\mathscr{L}_j$ be a relation over policies such that

$x\mathscr{L}_j y$ *IFF* For all $W(v_j)$, $FEAS(S(W(v_j),x)) = FEAS(S(W(v_j),y))$.

The reader can verify that $\mathscr{L}_j$ is an equivalence relation. Informally, $x\mathscr{L}_j y$ means that policy x and policy y behave equivalently (with respect to feasibility) under any sequence of applications of functions in Z. Let $\mathscr{L}_j(x)$ denote the equivalence class of x. The following procedure is suited to the search for any feasible policy.

4.4 *Procedure*: Upon MESSAGE(i,j,x), v_j computes $z_j(x)$ and determines $\mathscr{L}_j(z_j(x))$. If v_j has previously sent $z_j(y)$ to its neighbors, for some $z_j(x)\mathscr{L}_j z_j(y)$, then v_j sends no messages. Otherwise procedure 4.3 is invoked.

4.5 *Lemma*: Procedure 4.4 generates a solution to *P* $(R=\infty)$.
Proof. Let z be a solution to P, $z = S(W_1(v_*),\delta)$. Consider tracing W_1 versus an execution of procedure 4.4. Notice that any execution of 4.4 generates at least one prefix of W_1 because the initiator v_* sends δ to all its neighbors. Let W_2 be the longest prefix of W_1 generated by some execution of 4.4, where W_2 terminates at $v_{i(1)}$ Since W_2 is the longest prefix we infer that $v_{i(1)}$ did not send $x = S(W_2(v_*),\delta)$ to its neighbors. It follows that $v_{i(1)}$ previously sent y, for some $y\mathscr{L}_j x$. If x can be extended to feasibility, so can y, and we therefore continue tracing W_1 starting at $v_{i(1)}$ with policy y. This argument can repeated to exhaust W_1 and obtain a feasible solution. ■

Returning to the case R finite, cost is of importance. We now define a cost-sensitive equivalence relation similar to one defined in [3]. Let $\equiv_j$ be a relation over policies.
$x\equiv_j y$ *IFF*
(a) $\equiv_j$ is an equivalence relation,
(b) $x\mathscr{L}_j y$ holds, and
(c) for all $W(v_j)$, $COST(S(W(v_j),x))\leq COST(S(W(v_j),y))$.

4.6 *Procedure*: Upon MESSAGE(i,j,x), v_j computes $z_j(x)$ and determines $\equiv_j(z_j(x))$. If v_j has previously sent $z_j(y)$ to its neighbors, for some $z_j(x)\equiv_j z_j(y)$, and $COST(z_j(y))\leq COST(z_j(x))$, then v_j sends no messages. Otherwise procedure 4.3 is invoked.

4.7 *Lemma*: Procedure 4.6 generates a solution to P.
The proof is similar to the proof of lemma 4.5. Given some execution of procedure 4.6 and some optimum policy z, we can trace the walk of z and show that procedure 4.6 either generates z or another feasible policy of equal cost. ■

4.8 *Theorem*: There is an algorithm to solve P if

(a) Proposition (3.2) is satisfied, and
(b) $\equiv_j$ has finite rank for each v_j, and
(c) For all $v_j \in V$ and $x \in$ POLICY and for all $W(v_j)$, $COST(x) \leq COST(S(W(v_j),x))$.

Proof. There are three parts to the proof. First we show that any policy generated by procedure 4.6 is finite under conditions (a-c). This result will show termination of procedure 4.6. Finally, we appeal to previous work on diffusing computation [1,2] to detect termination and output a result.

(1) Any policy $S(W(v_*),\delta)$, generated by procedure 4.6, must satisfy: For all $v_j \in V$, $W(v_*)$ can contain at most M_j occurrences of v_j, where M_j denotes the rank of $\equiv_j$.

Proof (by contradiction). Suppose, on the contrary, that $W(v_*)$ contains $k > M_j$ occurrences of v_j, for some j. Let $W_1, \ldots, W_k$ be the prefixes of $W(v_*)$ that terminate at v_j. Clearly, there must be two distinct prefixes $W_{i(1)}$ and $W_{i(1)}$, "walks of the same equivalence class,"

$[S(W_{i(1)}(v_*),\delta)] \equiv_j [S(W_{i(2)}(v_*),\delta)]$.

Since $W_{i(1)}$ is a prefix of $W_{i(2)}$ (or vice-versa), we must conclude that

$COST[z_j(S(W_{i(2)}(v_*),\delta))] < COST[z_j(S(W_{i(1)}(v_*),\delta))]$

for if the walk $W_{i(2)}$ is extended, then by the logic of procedure 4.6, it must represent a lower cost policy than that of $W_{i(1)}$. But the equivalence of these two policies and part (c) of the premise implies the contrary, hence there is a contradiction.

(2) Procedure 4.6 terminates.
Proof. We prove termination by showing that every process v_j eventually reaches a permanently inactive state. Part (1) implies that every walk induced by an execution of procedure 4.6 has finite length. Since every step of the computation of procedure 4.6 extends or terminates some walk, and every walk is finite, we conclude that computations will cease in finite time.

(3) Termination detection/answer extraction.
Our plan to base an algorithm on procedure 4.6 will entail local variables for each process: Each v_j maintains a representative policy for each class of $\equiv_j$. The value of such a local variable is initially $\perp$, and is subsequently updated whenever MESSAGE(i,j,x) reflects an improvement in cost for $\equiv_j(z_j(x))$. Then the algorithm can succeed in two ways: First, some acceptable x may be found, in which case v_j discovers x and should broadcast a message throughout the distributed system to halt further activity. In the second instance, no feasible x with $COST(x) < R$ exists, so the algorithm produces a policy of optimum cost--which will reside in a local variable. This optimum policy must be extracted when the distributed computation halts; the diffusing computation protocols [1,2] provide suitable termination detection and extraction techniques. ■

5.0 *Applications*

5.1 *Tour*: This example is an approximation to the travelling salesman's problem. The problem is to find a low-cost tour through m vertices. For instance, if $m=3$, we wish to search the set of cycles r,p,q,r where p and q are distinct vertices. A motivation for this problem is that a process r may require a communication cycle through two other processes for the purpose of soliciting votes on major issues.

A policy will be a sequence of edges corresponding to some path beginning at v_*. In terms of proposition 3.2, a policy is feasible if it represents a cycle that begins and ends at v_*, contains m vertices, and has no repeated intermediate vertices. The cost of a policy is the sum of the weights of its edges.

The function z_j will extend policy MESSAGE(i,j,x) by adding (v_i,v_j) to x. Under this scheme, equivalence $x \equiv_j y$ holds if

(a) $z_j(x)$ and $z_j(y)$ cannot be extended to feasibility, that is, they contain repeated vertices or contain more than m vertices.
(b) x and y can be extended to feasibility, and they are both permutations of T, a subset of V, where $|T| \leq m$.

Following procedure 4.3, v_j will not send $z_j(x)$ to neighbors when case (a) applies. Case (b) implies that $\text{Rank}(\equiv_j) = 2^{|V|}$ in the worst case (m=n). This could lead to an exponential requirement for space, to accomodate local variables for each equivalence class. Since the travelling salesman problem is NP-complete, the exponential result is expected for a worst case.

5.2 *Shortest Walk*: Here we seek low-cost walks from the initiator to all other vertices. Edges have associated positive weights, and the cost of a walk is the sum of its edge-weights. A policy can be adequately represented by its cost and final vertex, since $x \equiv y$ holds for any walks x and y that terminate at v_j. Consequently $\text{Rank}(\equiv_j) = 1$ and v_j simply keeps track of the best policy reaching v_j during the course of the computation. This program is extended to the case of negative cycles in [2].

References

[1] E. W. Dijkstra and C. S. Scholten, Termination Detection for Diffusing Computations. *Information Processing Letters 11* 1(Aug 1980), pp. 1-4.
[2] K. M. Chandy and J. Misra, Shortest Path Algorithms, *Comm. ACM 25* 11(Nov 1982), pp. 833-837.
[3] R. M. Karp and M. Held, Finite-State Processes and Dynamic Programming, *Siam J. Appl. Math. 15* 3(May 1967), pp. 693-718.

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER	2. GOVT ACCESSION NO.	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) Manuscript: "A Paradigm for Detecting Quiescent Properties in Distributed Systems"		5. TYPE OF REPORT & PERIOD COVERED final: 6/14/81 - 6/15/85
		6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s) Professor K. M. Chandy Professor J. Misra University of Texas at Austin		8. CONTRACT OR GRANT NUMBER(s) AFOSR 81-0205
9. PERFORMING ORGANIZATION NAME AND ADDRESS Computer Sciences Department University of Texas at Austin Austin, Texas 78712		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
11. CONTROLLING OFFICE NAME AND ADDRESS Capt. A. L. Bellamy AFOSR/NM Bolling AFB, DC 20332		12. REPORT DATE July 1985
		13. NUMBER OF PAGES
14. MONITORING AGENCY NAME & ADDRESS(if different from Controlling Office)		15. SECURITY CLASS. (of this report)
		15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

NATO ASI Series, F13, Springer-Verlag Lecture Notes in Computer Science, to appear in 1985.

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

The problem of stability detection is one of the most widely studied problems in distributed computing. A stable property is one that persists: if the property holds at any point, then it holds thereafter. Examples of stable properties are termination, deadlock and loss of tokens in a token-ring. The problem is to devise algorithms to be superimposed on the underlying computation to determine whether a specified stable property holds for the underlying computation. This paper presents a simple (almost trivial) algorithm to detect quiescent properties, an important class of stable properties including those

mentioned above. Distributed snapshots may be used to derive algorithms for these problems. However our approach is this paper is different and results in simpler algorithms.

A PARADIGM FOR DETECTING QUIESCENT PROPERTIES IN DISTRIBUTED COMPUTATIONS*

K. Mani Chandy and Jayadev Misra
Department of Computer Sciences
University of Texas at Austin
Austin, Texas 78712

1. Introduction

The problem of stability detection is one of the most widely studied problems in distributed computing [1-28]. A *stable property* is one that persists: if the property holds at any point, then it holds thereafter. Examples of stable properties are termination, deadlock and loss of tokens in a token-ring. The problem is to devise algorithms to be superimposed on the underlying computation to determine whether a specified stable property holds for the underlying computation. This paper presents a simple (almost trivial) algorithm to detect *quiescent properties*, an important class of stable properties including those mentioned above. Distributed snapshots [7] may be used to derive algorithms for these problems. However our approach in this paper is different and results in simpler algorithms.

2. Model of Distributed Systems

2.1. The Model

A distributed system is a set of processes and a set of directed communication channels. Each channel is directed from one process to another process. Processes send messages on outgoing channels and receive messages on incoming channels. A process sends a message along an outgoing channel by depositing it in the channel. A process receives a message along an incoming channel by removing the message from the channel. A process may receive a message some arbitrary time after it is sent. Initially, all channels are *empty*. At any time each process is in one of a set of process states and each channel is in one of a set of channel states.

*This work was supported in part by a grant from the Air Force Office of Scientific Research under AFOSR 810205.

The channel state for a first-in-first-out channel is the sequence of messages in transit along the channel. For channels which deliver messages in arbitrary order, the channel state is the set of messages in transit. A system has a set of states, an initial state from this set, and a set of state transitions. The system state at any time is the set of process and channel states. Let S, S^* be states of a system. S^* is *reachable* from S if and only if there exists a sequence of state transitions from S to S^*. We assume that all system states are reachable from the initial system state.

2.2. Quiescent Property

A *stable property* B of a distributed system is a predicate on system states such that for all S^* reachable from S:

$B(S)$ *implies* $B(S^*)$

In other words, once a stable property becomes *true* it remains *true*. A *quiescent* property of a distributed system is a special kind of stable property characterized by (1) a subset P^* of the set of processes, (2) for all processes p in P^*, a predicate b_p on the process states of p and (3) a subset C^* of the set of channels between processes in P^*. A process p in P^* cannot send messages along channels in C^* while b_p holds. Furthermore, if b_p is *true*, it must remain *true* at least until p receives a message along a channel in C^*. The quiescent property B is:

all channels in C^* are *empty* and for all processes p in P^*: b_p.

It is easily seen that B is also a stable property. A process p is a *predecessor* of a process q with respect to B if and only if p and q are both in P^* and there exists a channel in C^* from p to q. For brevity we shall say p is a predecessor of q and drop the phrase "with respect to B". If for some system state, we have for some process q in P^*:

b_q *and* all of q's incoming channels in C^* are *empty and*

for all predecessors p of q: b_p (1)

then this condition must persist at least until for some predecessor p of q, b_p

becomes *false*. This fact is useful in understanding quiescent properties and their detection.

2.3. Problem Definition

Let the system computation go through a sequence of global states S_i, $i \geq 0$, where S_0 is the initial state; this sequence of global states will be called the *underlying computation*. Given a quiescent property B we wish to superimpose a detection algorithm on the underlying computation to determine whether B holds. The detection algorithm sets a boolean variable *claim* to *true* when it detects that B holds, and *claim* is *false* until that point. The detection algorithm must guarantee:

(Safety) : *not claim or B*

(Liveness) : within finite time of B becoming *true, claim* is set to *true.*

We now present a brief discussion of three instances of quiescent properties: termination, database deadlock and communication deadlock.

2.4. Termination

A computation is defined to be *terminated* if and only if all processes are *idle* and all channels are *empty*. Thus C^* is the set of all channels, P^* is the set of all processes, and for each process *p*, b_p is: *p* is *idle*. Idle processes don't send messages and hence termination is a quiescent property.

2.5. Database Deadlock

A process is either *active* or *waiting*. A waiting state of a process *p* is specified by a pair (R_p, H_p) where R_p is a non-empty set of resources that *p* is waiting for and H_p is a set of resources that *p* needs and holds (where R_p and H_p have no common elements). Resources are sent as messages from active processes to other processes; a waiting process does not send any resource it needs and holds. A process *p*, in a waiting state specified by (R_p, H_p), takes the following action on receiving a resource *r* in R_p:

```
begin  $R_p := R_p - \{r\}$;  $H_p := H_p \cup \{r\}$;
        if $R_p = \{\ \}$ then become active else wait
end
```

Here { } is the empty set. When p transits from active to waiting state, R_p and H_p are set to values which are of no consequence to us here. A set P^* of processes is deadlocked if every process in P^* is waiting for resources held by other processes in P^*, i.e.

P^* is database deadlocked $\equiv$

for all p in P^*: p is waiting and there exists a q in P^* such that

$R_p \cap H_q \neq \{\ \}$

In this case, the predicate b_p is: p is waiting for R_p and p holds H_p. A channel c is in C^* if and only if c is from a process q to a process p where p and q are both in P^*, and q holds a resource required by p. Typically, P^* is not specified and it is required to obtain a P^* as part of the detection algorithm.

2.6. Communication Deadlock

As in database deadlock a process is *active* or *waiting*. A waiting process p is waiting on a set of incoming channels C_p; on receiving a message along *any* channel in C_p, process p becomes active. An active process may start waiting at any time. Until it receives a message along a channel in C_p, a waiting process p continues to wait on C_p. A waiting process cannot send messages. A set of waiting processes is deadlocked if no process in the set is waiting on a channel from a process outside the set, and all channels between processes in the set are *empty*, i.e.,

A set of processes P^* is communication deadlocked $\equiv$

for all p in P^*: p is waiting for a set of incoming channels C_p where each channel c in C_p is from a process in P^*, and c is *empty*.

In this case, b_p is: p is waiting on C_p. C^* is the union of all C_p for p in P^*.

As in database deadlock, the detection algorithm is required to find P^* if such a set exists. Next we consider two specific classes of distributed systems: (i) systems in which messages are acknowledged and (ii) systems in which channels are first-in-first-out, and show how to detect quiescent properties in each class. The latter class needs little description. We describe the former class next.

2.7. Systems with Acknowledgements

Let c be a channel from a process p to a process q. On receiving a message along c, process q sends an acknowledgement ack_c to p. We are not concerned with how *acks* travel from one process to another. An *ack* is not considered to be a message in that *acks* are not acknowledged in turn. Furthermore, the statement "channel c is *empty*" means that c contains no message; it may or may not contain *acks*. Let num_c be the number of unacknowledged messages p has sent along outgoing channel c, i.e.,

$num_c =$ number of messages sent by p along c —
number of ack_c acknowledgements received by p.

$num_c = 0$ *implies* c is *empty*.

We assume that every message sent is received in finite time and acknowledged in finite time. We also assume that every *ack* sent is received in finite time. Hence, an acknowledgement is received for each message within finite time of sending the message. Therefore,

if B becomes *true*, then within finite time of B becoming *true*:

for all c in C^*: $num_c = 0$

3. The Paradigm

Our paradigm is based on observing each process computation for some period of time called an observation period. An *observation period* for a process p is specified by two integers, $start_p$ and end_p, $start_p \leq end_p$, denoting that p's computation is observed at every S_i, $start_p \leq i \leq end_p$. An *observation period set* for a quiescent property B is a set of observation periods, one for each process in P^*.

An observation period set $obs'' = \{(start_p'', end_p'') \mid p \text{ in } P^*\}$ *is later than* an observation period set $obs' = \{(start_p', end_p') \mid p \text{ in } P^*\}$ if and only if all starting times in obs'' are after some starting time in obs', i.e.

$$\min_p \; start_p'' \; > \; \min_p \; start_p'$$

Let B^* be a predicate on observation period sets, defined as follows.

$B^*(obs) \equiv$ [for all p in P^* :

for all states S_i where $start_p \leq i \leq end_p$: b_p holds in S_i]

and

[for all p,q in P^* where p is a predecessor of q: all messages sent

by p at or before $start_p$ are received by q at or before end_q] (2)

Note: To ensure that messages sent by p at or before $start_p$ are received by q at or before end_q, we must have for all p,q in P^* where p is a predecessor of p : $start_p \leq end_q$ (3)

3.1. Quiescence Detection Paradigm

```
claim := false; obtain an observation period set obs;
while not $B^*(obs)$ do
    obtain an observation period set obs' later than obs;
    obs := obs'
od;
claim := true
```

We next prove the correctness of this paradigm and postpone discussion of techniques for implementing the paradigm to a later section.

3.2. Proof of Correctness

Safety : *not claim or B*

Safety holds while *claim* is *false*; therefore consider the final iteration of the while loop after which *claim* is set to *true*. For this iteration, we prove the following by inducting on i:

for all $i \geq 0$: for all p in P^* :

[[$i < start_p$ or b_p holds in S_i] *and*

[$i < end_p$ or p's incoming channels in c are empty]]

This induction follows from (1), (2), and (3).

Liveness: If there exists an $i \geq 0$ such that B holds for S_i then there exists a $j \geq 0$ such that *claim* = *true* in S_j . If B holds for S_i then for all observation period sets, *obs*, where $start_p \geq i$, for all p, B^* (*obs*) holds. From the paradigm, either *claim* is set *true* or later observation periods are chosen indefinitely. Hence if B holds for S_i for any $i \geq 0$, then *claim* will be *true* for some S_j, $j \geq 0$.

3.3. Implementation of the Paradigm

The key question for implementation is: How can we ensure that all messages sent by a predecessor p of a process q at or before $start_p$, are received by q at or before end_q?

3.3.1. Systems with Acknowledgements

The above question can be answered for systems with acknowledgements by ensuring the following condition: for all p,q in P^* where p is a predecessor of q and for all channels c from p to q :

$num_c{=}0$ at $start_p$ *and* $start_p \leq end_q$.

Proof of this condition is as follows. At $start_p$, $num_c{=}0$ implies that c is empty and hence all messages sent along c have been received. Hence all messages sent at or before $start_p$, along c, are received at or before $start_p$ and since $start_p \leq end_q$, the result follows.

For all p in P^*, let $quiet_p \equiv$ for all states S_i, where $start_p \leq i \leq end_p$: [b_p and for all outgoing channels c in C^* : $num_c = 0$].

In the paradigm we replace B^* (*obs*) by [for all p in P^* : $quiet_p$] *and*

[for all p,q in P^* where p is a predecessor of q : $start_p \leq end_q$].

We show, in section 4, how $start_p \leq end_q$, can be maintained. num_c is maintained as a local variable of p and hence $quiet_p$ can be determined by p. Note that for systems with rendezvous, such as CSP and ADA, num_c=0 holds at all times.

3.3.2. Systems with First-In-First-Out Channels

To answer the key question posed at the beginning of this section, we use special messages called *markers*, which are sent and received along channels in C^*. They have no effect on the underlying computation other than that they occupy the same channels as regular messages. We use the following implementation rules.

R1. Every process p in P^* sends one marker along each outgoing channel in C^* some (finite) time after (or at) $start_p$ and,

R2. Every process p in P^* has received one marker along each incoming channel in C^* some time before (or at) end_p.

Since channels are first-in-first-out, all messages sent along a channel before the marker is sent on the channel must be received before the marker is received. Hence every message sent at or before $start_p$ is received at or before end_q, for all p,q in P^*, where p is a predecessor of q.

Each process p in P^*maintains a local boolean variable $quiet_p$ where

$$quiet_p \equiv \text{ for all states } S_i \text{ where } start_p \leq i \leq end_p : b_p.$$

In the paradigm we replace $B^*(obs)$ by : [for all p in P^* : $quiet_p$] and rules **R1**, **R2** are satisfied.

3.4. Notes on the Paradigm

Our constraints on observation period sets are weak. For instance it is possible that for a predecessor p of q, $start_q > end_p$ and there may be *no overlap* between

p's and q's observation periods. For a system with first-in-first-out channels, process p may send markers on some or all outgoing channels *after* end_p, and may receive markers on some or all incoming channels *before* $start_p$.

If the quiescent property never holds, the iteration in the paradigm will never terminate, i.e. an infinite sequence of observation period sets will be obtained.

4. Applications of the Paradigm

There are many problems to which the paradigm may be applied and many ways of applying the paradigm. We show two examples to demonstrate the power of the paradigm: termination detection and (both types of) deadlock detection, described earlier. We use termination detection as an example of the use of markers and deadlock detection as an example of the use of acks.

4.1. Termination Detection

Processes are labeled p_i, $0 \leq i < n$. We employ a *token* to transmit the values $quiet_p$. The token cycles through the processes visiting $p_{(i+1) mod\, n}$ after departing from p_i, all i. A cycle is initiated by a process p_{init}, called the initiator. If the token completes a cycle (i.e. returns to p_{init} after visiting all processes) and if all processes p return a value $quiet_p$ of *true* in this cycle then the initiator detects termination, i.e. it sets *claim* to *true*. If any process q returns a value $quiet_q$ of *false* in a cycle, then the current cycle is terminated and a new cycle is initiated with q as the initiator. A process ends one observation period and immediately starts the next observation period when it sends the token. The algorithm, described next in detail, shows how $quiet_p$ is set.

4.1.1. The Algorithm

The are no shared variables in a distributed system. However, *for purposes of exposition* we assume that *claim* is a shared global variable which has an initial value of *false* and which may be set *true* by any process. Such a global variable can be simulated by message transmissions; for instance, the process that sets *claim* to *true* may send messages to all other processes notifying them.

Two types of messages are employed in the termination detection algorithm.

<marker> : this type of message has already been discussed; it carries no other information (except its own type).

<token, initiator> : this is the token and its initiator, as described in Section 4·1.

Each process has the following constants and variables. These will be subscripted, by i, when referring to a specific process i.

ic: number of incoming channels to the process, a constant,

idle: process is idle,

quiet: process has been continuously idle since the token was last sent by the process; *false* if the token has never been sent by this process,

hold-token: process holds the token,

init: the value of initiator in the <token, initiator> message last sent or received; undefined if the process has never received such a message,

m: number of markers received, since the token was last sent by the process; initial value as given in the algorithm.

Initial Conditions

The token is at p_0.

m_i = the number of channels from processes with indices greater than i, for all i, i.e., the cardinality of the set, $\{c \mid c \text{ is a channel from } p_j \text{ to } p_i \textit{ and } j > i\}$.

(This initial condition is required because otherwise, the token will permanently stay at one process.)

$quiet_i = false$, for all i.

(The algorithm is slightly more efficient with different initial conditions, but for purposes of exposition we shall make the simpler assumption.)

$$hold\text{-}token_i = \begin{cases} true, \text{ for } i=0 \\ false, \text{ for } i \neq 0 \end{cases}$$

$init_i$ is arbitrary, for all i

Algorithm for a Process P_i

The algorithm for a process is a repetitive guarded command. The repetitive guarded command is a set of rules where each rule is of the form, *condition* $\rightarrow$ *action*. The algorithm proceeds as follows: one of the rules whose condition part evaluates to *true* is selected nondeterministically and its action part is executed. The repetitive guarded command consists of the following rules:

```
1. receive marker  → $m_i := m_i + 1$;

2. $quiet_i$ and receive regular message (i.e. underlying computation's
   message) →  $quiet_i := false$;

3. receive <token, initiator>  →  begin $init_i := initiator$; $hold\text{-}token_i$
   $:= true$ end;

4. $hold\text{-}token_i$ and $(ic_i = m_i)$ and $idle_i$ →
if $quiet_i$ and $(init_i = i)$ then {termination detected} $claim := true$;

if $quiet_i$ and $(init_i \neq i)$ then {continue old cycle}

                    begin

                         $m_i := 0$ ;

                         Send marker along each outgoing channel;

                         $hold\text{-}token_i := false$;

                         send <token, $init_i$> to $p_{(i+1) \bmod n}$

                    end
```

```
if ~$quiet_i$          then {initiate new cycle}
                       begin
                           $m_i := 0$; $quiet_i := true$; $init_i := i$;
                           Send marker along each outgoing channel;
                           $hold\text{-}token_i := false$;
                           send $<token, init_i>$ to $p_{(i+1) mod\ n}$
                       end
```

4.1.2. Proof of Correctness

We need merely show that the algorithm fits the paradigm. A process p_i ends an observation period and starts the next one when the token leaves p_i. Initially, an observation period is started when the token leaves p_0; the values of m_i are so chosen initially that it is possible for the token to leave p_i, for the first time, when p_i has received markers from all lower numbered processes. We need to show the following {initial conditions should be treated slightly differently}:

1. $quiet_i \equiv$ p_i has been continuously idle in the current observation period, i.e. since the token last left p_i
2. Each process sends a marker on each outgoing channel upon starting an observation period.
3. Each process ends an observation period only after receiving exactly one marker along each incoming channel.
4. *claim* is set to *true* if and only if in one cycle of the token (which corresponds to an iteration of the paradigm all processes p_i return a value of $quiet_i = true$ at the end of their observation periods.
5. After termination, a cycle of the token is completed in finite time. To guarantee this we must ensure that each process receives a marker along each incoming channel in finite time.

Proofs of these assertions follow directly from the algorithm and the details are left to the reader.

4.1.3. Overhead and Efficiency

The most overhead is incurred in rule 4, when a process is idle. The overhead while a process is doing useful work in negligible. Also a process sends the token only when the process is idle; this controls the rate at which the token cycles through processes. For instance, if all processes are active, the token will not move at all. Also observe that termination will be detected within two cycles of after computation terminates.

4.2. Deadlock Detection

The following refinement of the paradigm is applicable to database deadlock and communication deadlock, under the assumption that messages are acknowledged.

A process which we call the *detector* sends *initiate* messages to all processes; on receiving an *initiate* message a process starts its observation period and acknowledges the *initiate* message. After receiving acknowledgements to all the *initiate* messages sent the detector sends *finish* messages to all processes. A process p ends its observation period after receiving a *finish* message and replies with a boolean value $quiet_p$ and a set $waiting\text{-}for_p$, where

$$quiet_p \equiv \text{ for all states in the observation period : } [p \text{ is } waiting \text{ } and \text{ for all outgoing channels } c : num_c = 0]$$

$$waiting\text{-}for_p = \begin{cases} \text{set of objects that } p \text{ is waiting for in the observation period, if } quiet_p. \\ \text{arbitrary, if } not \ quiet_p \end{cases}$$

The *detector* determines whether there exists a set of processes P^*, such that for all p in P^* : $quiet_p$ and the sets $waiting\text{-}for_p$ are such as to constitute a deadlock. The proof of correctness is that the algorithm fits the paradigm.

The algorithm, as stated above, appears to be centralized rather than distributed. Note however, that the *detector* process could be different for different initiations and there could be multiple *detectors*. The function of the detector, i.e. sending messages, detecting deadlock, can be decentralized by having messages forwarded to their destinations by intermediate processes and deadlock detection computation carried out by intermediate processes.

5. Previous Work

The idea of observation periods is central to the works of Francez, Rodeh and Sintzoff on distributed termination [12-14], and Chandy, Misra and Haas on deadlock detection [6]. Dijkstra [11], Gouda [16] and Misra [26] have developed token based algorithms for termination detection, and these algorithms also use observations over a period. We have attempted to generalize these works to produce a particularly simple paradigm for detecting an important class of properties, quiescent properties, in distributed systems with asynchronous channels.

Acknowledgement: It is a pleasure to acknowledge extensive discussions with Shmuel Katz, who also carefully read and commented on earlier draft of this paper. Edsger W. Dijkstra and Hank Korth helped with constructive criticism.

References

1. C. Beeri and R. Obermarck, "A Resource Class Independent Deadlock Detection Algorithm", *Research Report RJ3077,* IBM Research Laboratory, San Jose, California, May 1981.

2. G. Bracha and S. Toueg, "A Distributed Algorithm For Generalized Deadlock Detection", *Technical Report TR 83-558,* Cornell University, June 1983.

3. K. M. Chandy and J. Misra, "Asynchronous Distributed Simulation Via a Sequence of Parallel Computations", *Communications of the ACM,* Vol. 24, No. 4, pp. 198-205, April 1981.

4. K. M. Chandy and J. Misra, "A Distributed Algorithm for Detecting Resource Deadlocks in Distributed Systems", *ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing,* Ottawa, Canada, August 1982.

5. K. M. Chandy and J. Misra, "A Computation on Graphs: Shortest Path Algorithms", *Communications of the ACM,* Vol. 25, No. 11, pp. 833-837, November 1982.

6. K. M. Chandy and J. Misra and L. Haas, "Distributed Deadlock Detection", *ACM Transactions on Computing Systems,* Vol. 1, No. 2, pp. 144-156, May 1983.

7. K. M. Chandy and L. Lamport, "Distributed Snapshots: Determining Global States of Distributed Systems", to appear in *ACM Transactions on Computing Systems.*

8. E. Chang, "Echo Algorithms: Depth Parallel Operations on General Graphs", *IEEE Transactions on Software Engineering,* Vol. SE-8, No. 4, pp. 391-401, July 1982.

9. S. Cohen and D. Lehmann, "Dynamic Systems and Their Distributed Termination", *ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing,* pp. 29-33, Ottawa, Canada, August 18-20, 1982.

10. E. W. Dijkstra and C. S. Scholten, "Termination Detection for Diffusing Computations", *Information Processing Letters,* Vol. 11, No. 1, August 1980.

11. E. W. Dijkstra, "Distributed Termination Detection Revisited", EWD 828, Plataanstraat 5, 5671 AL Nuenen, The Netherlands.

12. N. Francez, "Distributed Termination", *ACM Transactions on Programming Languages and Systems,* Vol. 2, No. 1, pp. 42-55, January 1980.

13. N. Francez, M. Rodeh, and M. Sintzoff, "Distributed Termination with Interval Assertions", *Proceedings of Formalization of Programming Concepts,* Peninusla, Spain, April 1981. Lecture Notes in Computer Science 107, (Springer-Verlag).

14. N. Francez and M. Rodeh, "Achieving Distributed Termination Without Freezing", *IEEE-TSE,* Vol. SE-8, No. 3, pp. 287-292, May 1982.

15. V. Gligor and S. Shattuck, "On Deadlock Detection in Distributed Data Bases", *IEEE Transactions on Software Engineering,* Vol. SE-6, No. 5, September 1980.

16. M. Gouda, "Personal Communication", Department of Computer Sciences, University of Texas, Austin, Texas 78712.

17. L. Haas and C. Mohan, "A Distributed Deadlock Detection Algorithm for a Resource-Based System", *Research Report RJ3765,* IBM Research Laboratory, San Jose, California, January 1983.

18. T. Herman and K. M. Chandy, "A Distributed Procedure to Detect AND/OR Deadlock", Computer Sciences Department, University of Texas, Austin, Texas 78712, February 1983.

19. T. Holt, "Some Deadlock Properties of Computer Systems", *Computing Surveys,* Vol. 4, No. 3, pp. 179-196, September 1972.

20. D. Kumar, Ph.D Theses (in preparation), Computer Sciences Department, University of Texas, Austin, Texas 78712.

21. L. Lamport, "Time, Clocks and the Ordering of Events in a Distributed System", *Communications of the ACM,* Vol. 21, No. 7, July 1978.

22. G. Le Lann, "Distributed Systems - Towards a Formal Approach", *Information Processing 77,* IFIP, North-Holland Publishing Company, 1977.

23. D. Menasce and R. Muntz, "Locking and Deadlock Detection in Distributed Data Bases", *IEEE Transactions on Software Engineering,* Vol. SE-5, No. 3, May 1979.

24. J. Misra and K. M. Chandy, "A Distributed Graph Algorithm: Knot Detection", *ACM Transactions on Programming Languages and Systems,* Vol. 4, No. 4, pp. 678-688, October 1982.

25. J. Misra and K. M. Chandy, "Termination Detection of Diffusing Computations in Communicating Sequential Processes", *ACM Transactions on Programming Languages and Systems,* Vol. 4, No. 1, pp. 37-43, January 1982.

26. J. Misra, "Detecting Termination of Distributed Computations Using Markers", *Proceedings of the ACM SIGACT-SIGOPS Symposium of Principles of Distributed Computing,* Montreal, Canada, August 17 - 19, 1983.

27. R. Obermarck, "Deadlock Detection For All Resource Classes", *Research Report RJ2055,* IBM Research Laboratory, San Jose, California, October 1980.

28. R. Obermarck, "Distributed Deadlock Detection Algorithm", *ACM Transactions on Database Systems,* Vol. 7, No. 2, pp. 187-208, June 1982.

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER	2. GOVT ACCESSION NO.	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) Manuscript: "An Example of Stepwise Refinement of Distributed Programs: Quiescence Detection"		5. TYPE OF REPORT & PERIOD COVERED final: 6/14/81 - 6/15/85
		6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s) Professor K. M. Chandy Professor J. Misra The University of Texas at Austin		8. CONTRACT OR GRANT NUMBER(s) AFOSR 81-0205
9. PERFORMING ORGANIZATION NAME AND ADDRESS Computer Sciences Department University of Texas at Austin Austin, Texas 78712		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
11. CONTROLLING OFFICE NAME AND ADDRESS Capt. A. L. Bellamy AFOSR/NM Bolling AFB, DC 20332		12. REPORT DATE July 1985
		13. NUMBER OF PAGES
14. MONITORING AGENCY NAME & ADDRESS(if different from Controlling Office)		15. SECURITY CLASS. (of this report)
		15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

to appear in ACM Transactions on Programming Languages and Systems

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

This paper presents a systematic derivation of a distributed algorithm by a sequence of successive refinements of its specification: the invariant and termination conditions. The problem treated is detection of "quiescent properties" in a distributed computation. Such properties include termination and deadlock. An argument is made that stepwise refinement is a useful method to derive distributed programs.

An Example of Stepwise Refinement of Distributed Programs: Quiescence Detection

K. Mani Chandy

Jayadev Misra

Department of Computer Sciences

University of Texas at Austin

Austin, Texas, 78712

29 April 1985

This work was supported in part by a grant from the Air Force Office of Scientific Research under AFOSR 810205.

1. Introduction

This paper presents a systematic derivation of a distributed algorithm by a sequence of successive refinements of its specification: the invariant and termination conditions. The problem treated is detection of "quiescent properties" in a distributed computation. Such properties include termination and deadlock. An argument is made that stepwise refinement is a useful method to derive distributed programs.

A consequence of the stepwise refinement method is that we obtain, in a systematic manner, the weakest conditions under which the algorithm can operate. Specific instances of the quiescence detection problem have been extensively studied [1-30]. All such algorithms have the feature that each process is observed over some interval during the computation and the intervals are related in some manner. For instance, in termination detections of diffusing computations [10], messages define the beginning and acknowledgements define the end of an interval. A token is often used [14,28] whose successive receipts at a process define various intervals. Our solution differs from these in that we inspect processes at *arbitrary times* and in *arbitrary order*.

Stepwise refinements in our distributed program is carried out as follows: the problem to be solved is specified in terms of global (system wide) properties; a refinement consists of generalizing this global property to apply to subsystems. For instance, quiescence is a system wide property; refinement consists of obtaining generalizations of quiescence which apply to subsystems. Generalization of the desired properties results in weakening the invariant. The weakened invariant suggests the structure of the desired algorithm.

We use a common model of distributed systems in which sends and receives are asynchronous, and channels are error-free [7]. A distributed system is a set of processes and a set of directed channels. Each channel is directed from one process to another. A process sends messages on its outgoing channels and receives messages on its incoming channels. Messages are delayed for arbitrary, finite times in channels. Channels are first-in-first-out. Processes communicate with one another exclusively by sending/receiving messages. (For a more formal description see [7].)

Each process and each message is colored either *black* or *white*. A message is given the color of the process sending it: *black* processes send *black* messages and *white* processes send *white* messages. The color of a message does not change. A *black* process may turn *white* at any time. A *white* process may turn *black* only upon receipt of a *black* message. The problem is to detect property W where

$$W \equiv \text{all processes and messages in the system are } \textit{white}. \quad (1)$$

Once W holds it continues to hold. Property W is called *quiescence*. "Many distributed algorithms are structured as a sequence of phases where each phase consists of a transient part followed by a stable part . . . The presence of stable behavior indicates the end of a phase" [3,7]. States in which there is a *black* process or message are in the transient part, and states in which all processes and messages are *white* are in the stable part. Thus detecting W amounts to detecting the end of a phase. Termination and deadlock are special cases of W. In deadlock detection "*white* process" means "waiting process", and a *black* message is one which causes its receiver to stop waiting. Similar definitions apply to termination detection.

The detection algorithm is to be *superimposed* on the underlying computation. A superimposed algorithm does not alter the underlying computation. The superimposed algorithm employs the processes and channels of the underlying computation. However, the superimposed algorithm may use additional local variables at each process and special messages which are not part of the underlying computation. Actions that change the state of the underlying computation may also change the state of local variables employed by the superimposed computation. There may also be actions that change the state of the superimposed computation and do not change the state of the underlying computation.

1.1. Specification of the Detection Algorithm

The detection algorithm has a boolean variable *claim* satisfying the following invariant and termination conditions.

Invariant: *not claim or W* (2)

Termination: within finite time of W holding, *claim* holds (3)

Therefore the detection algorithm is required to set *claim* to *true* only if W holds and it must do so within finite time of W being true.

1.2. An Outline of the Paradigm

We first discuss the paradigm informally and then discuss how the specifications are met starting with the invariant (2) and then considering the termination condition (3). We use a set *checked* of processes such that the "algorithm detects W" means all processes are in *checked.* Formally,

$$claim \equiv (checked = P) \tag{4}$$

where P is the set of all processes. Let *unchecked* be the complement of *checked* i.e. *unchecked* $=$ P - *checked.* For brevity, processes in *checked* are called *checked* processes; processes in *unchecked* are called *vnchecked* processes. We wish to develop an algorithm which has two basic actions (1) add an *unchecked* process to set *checked* and (2) remove processes from *checked.* The algorithm is developed so that all processes are in *checked* means *claim* holds. Now conditions (2, 3) can be rewritten:

Invariant: $(checked \subset P)$ *or* W (5)

Termination: Within finite time of W holding, $checked = P$. (6)

Invariant (5) means there is at least one *unchecked* process *or* W holds. The termination condition is that within finite time of W holding all processes are in *checked.* Property W is a system wide property. Following Dijkstra [11] and Gries [18], we seek to weaken (5, 6) by replacing the system wide property W by a subsystem property w defined on process sets S, $S \subseteq P$, such that

$$w(P) = W.$$

From $w(P) = W$, we obtain:

$(\mathit{checked} \subset P)$ *or* $W \equiv (\mathit{checked} \subset P)$ *or* $w(\mathit{checked})$

The above equivalence follows by considering two cases: (1) $\mathit{checked} \subset P$ and (2) $\mathit{checked} = P$. In the former case, the first term in the disjunctions of both the left hand side and the right hand side of the equivalence hold. In the second case we have $w(\mathit{checked}) \equiv w(P)$ and $w(P) \equiv W$; hence the second term of the disjunctions of the left and right hand sides are equivalent.

This allows us to write (5, 6) as:

Invariant: $(\mathit{checked} \subset P)$ *or* $w(\mathit{checked})$ (7)

Termination: Within finite time of $w(P)$ holding, $\mathit{checked} = P$ (8)

How shall we generalize system-wide property W to obtain a subsystem property w? A definition of w which guarantees $w(P) = W$ is:

$w(S) \equiv$ all processes in S are *white and*
all input channels of all processes in S contain only *white* messages.

Invariant (7) says that there is at least one *unchecked* process or all processes in *checked* are *white* and have input channels containing no *black* message.

1.3. How the Algorithm Maintains Invariant (7)

1.3.1. Intuition

We first discuss the intuition which went into the development of the algorithm so as to maintain invariant (7). Initially, (7) holds by setting *checked* = *empty*. To maintain invariant (7) it follows that if *w(checked)* does not hold then we should not allow *checked* to equal P. In other words, if *w(checked)* does not hold we must prevent at least one *unchecked* process from being added to *checked*. How do we prevent this?

We postulate an inclusion condition, *inc*, such that a process is added to *checked* only if it satisfies *inc*. We define *inc* so that the following is an invariant:

Invariant: *w(checked) or* for some *unchecked* process: *not inc* (9)

Invariant (9) assures us that if $w(\mathit{checked})$ does not hold then $\mathit{checked} \subset P$ because there is at least one *unchecked* process which does not satisfy *inc*; furthermore this *unchecked* process cannot be added to *checked* and thus we maintain the invariant. We shall devise an algorithm satisfying invariant (9) and thus ensure invariant (7). Now we look for a condition *inc* satisfying (9).

1.3.2. An Example of an Incorrect Inclusion Condition

A simplistic, but incorrect, inclusion condition for a process is: process is *white* and all its incoming channels contain only *white* messages. To see why this condition is incorrect consider a system with two processes. Suppose *checked* is *empty* and then one of the processes is added to *checked* when the other process is *black*. Then the *black* process sends a *black* message, turns *white* and is subsequently added to *checked*. Now we have a situation in which the algorithm reports (since $checked = P$) that all processes and messages are *white*, though there is a *black* message in transit. What went wrong? Invariant (9) was not satisfied. The inclusion condition was not strong enough to ensure that at least one *unchecked* process would never satisfy the inclusion condition if a *checked* process was sent a *black* message. We now give an example of an inclusion condition which does satisfy invariant (9). (There may be more than one inclusion condition satisfying (9) — we should choose one most appropriate to each problem.)

1.3.3. A Correct Inclusion Condition

We develop a correct inclusion condition based on the following observation. For any subset of processes *checked*, if $w(checked)$ holds then it continues to hold until (1) a *checked* process is sent a *black* message by an *unchecked* process or (2) an *unchecked* process is added to *checked* Let us first focus our attention on the event: an *unchecked* process sends a *black* message to a *checked* process.

We maintain invariant (9) as follows: if an *unchecked* process has sent a *black* message to a *checked* process then that *unchecked* process is not added to *checked*; we shall define the inclusion condition *inc* so that such an *unchecked* process does not satisfy *inc* and this ensures that such a process is not added to *checked*. Special messages called *markers* are employed; *markers* have no effect on the underlying computation. We will enforce the following: if an *unchecked* process sends a marker along a channel and subsequently sends a *black* message along that channel then that *unchecked* process does not satisfy the inclusion condition, *inc*.

Each process has a local variable, *channel-state* for each of its output channels. A *channel-state* has one of three values: *pre-marker, positive* or *negative.* A channel is initially in *pre-marker* state and transits from *pre-marker* state to *positive* state when a marker is sent along it; it transits from *positive* to *negative* when a *black* message is sent along it. Therefore, a channel is *positive* means that a *marker* has been sent along the channel and no *black* message has been sent along the channel after the *marker* along it. A channel is *negative* means that at least one *black* message has been sent along it after the *marker* was sent along it. The information conveyed by the *marker* is this: after a process receives a *marker* along a channel the following holds: the channel contains only *white* messages *or* the channel is *negative*

Each process has a boolean variable, *received-marker*, for each of its input channels where initially *received-marker* is *false*, and it becomes *true* when a *marker* is received along the channel. Observe that:

for all channels: *not* ((*channel-state* = *pre-marker*) *and received-marker*)

The meaning of a *marker* is given by the invariant:

for all channels: *not received-marker or* channel
contains only *white* messages *or channel-state* = *negative* (10)

The inclusion condition for a process is:

inc ≡ process is *white and*
[for all its input channels: *received-marker*] *and*
[for all its output channels: channel is *nonnegative*] (11)

In other words, we add a process to *checked* only it if is *white* and it has received a *marker* along each of its input channels and it has not sent a *black* message following the *marker*, along any output channel. Given (11), we prove the following invariant:

Invariant: *w(checked) and*
all output channels of all processes in *checked* are *nonnegative*
or
there is a *negative* output channel
from an *unchecked* process to a *checked* process. (12)

Note: (12) implies (9). Therefore it is sufficient to prove that (12) is invariant.

Lemma: Inclusion condition (11) maintains invariant (12).

Proof: We prove the invariant by induction on the cardinality of *checked.*

Initially, *checked* is empty; hence (12) holds. Assume that (12) holds at some point in the computation immediately before some *unchecked* process q is added to *checked*; we show that (12) holds with *checked* replaced by $\{q\} \cup$ *checked.*

If there is a *negative* channel from a process in *unchecked* $- \{q\}$ to a process in *checked* $\cup \{q\}$, then the invariant is maintained q is added to *checked.* Therefore, assume that

(a): all channels from *unchecked* $- \{q\}$ to *checked* $\cup \{q\}$ are *nonnegative*

From the inclusion condition (11), all output channels of q are *nonnegative.* Therefore using (a), we have,

(b): all channels from *unchecked* processes to *checked* processes are *nonnegative.*

From invariant (12) and (b) we have,

(c): *w(checked)* and all output channels of processes in *checked* are *nonnegative.*

From (a) all channels from *unchecked* $- \{q\}$ to q are *nonnegative*; from this fact and (c), we have,

(d): all input channels of q are *nonnegative.*

From inclusion condition (11)

(e): q is *white.*

From (10), (11) and (d) we have,

(f): all input channels of q contain only *white* messages.

From (c), (e) and (f) we have:

(*g*): $w(checked \cup \{q\})$.

From (*c*), all output channels of all process in checked are *nonnegative*; from inclusion condition (11), all output channels of q are *nonnegative*. Therefore, we have,

(*h*): all output channels of all processes in $checked \cup \{q\}$ are *nonnegative*.

Invariance of (12), with checked replaced by $checked \cup \{q\}$, follows from (*g*) and (*h*).

Next, we show that (12) is maintained when a message is sent or received; we prove only the case of message send and leave receives to the reader. If the second term in the disjunction (12) is true prior to a send, it remains true following the send. Hence assume that the second term is false and therefore the first term in the disjunction is true, prior to the message send. If the message is sent by a *checked* process, it must be *white* because $w(checked)$ holds prior to the send and hence all *checked* processes must be *white* before the send.

A message send can be (*a*) a message sent by an *unchecked* process to an *unchecked* process or (*b*) a *white* message sent by an *unchecked* process to a *checked* process or (*c*) a *black* message sent by an *unchecked* process to a *checked* process *or* (*d*) a *white* message sent by a *checked* process. In cases (*a*), (*b*), and (*d*), the first term of the disjunction (12) is not falsified by the send. In case (*c*) the second term of the disjunction holds after the send.

1.4. How The Algorithm Achieves Termination

1.4.1. Intuition

If an *unchecked* process has a *negative* output channel then *checked* can never equal *P*, from the inclusion condition. Hence termination will never be detected. To ensure that termination condition (8) is met, each *negative* channel is reinitialized in finite time, where by reinitialization we mean the *channel-state* is set to *pre-marker* state; in order to preserve (10), we set the corresponding *received-marker* value to *false*, and to preserve invariant (12), if the channel is from an *unchecked* process to a *checked* process then we set *checked* to *empty*. We now argue that if each *negative* channel is reinitialized in finite time the termination condition (8) is satisfied.

1.4.2. Proof of Termination

If *W* holds, all *nonnegative* channels are *positive* (and remain *positive*) or are in *pre-marker* state and will become *positive* in finite time (and remain *positive* thereafter). From our reinitialization procedure, a *negative* channel becomes *pre-marker* in finite time. Therefore, all channels are *positive* in finite time after *W* holds. All processes are *white*. Since *markers* are delivered in finite time, for all channels *received-marker* holds in finite time after the channel becomes *positive*. Therefore, in finite time after *W* holds, *inc* holds for every process, and hence, within finite time after *W checked* = *P*.

1.5. The Quiescence Detection Paradigm

Now, we put the pieces that we have been developing together to obtain the quiescence detection paradigm. Our description consists of a repetitive guarded command [11] which is a set of statements of the form *condition* → *action*. The *action* part of a statement is executed if the *condition* part of the statement holds. The repetitive guarded command terminates when the *condition* parts of all statements in the command are *false*.

In this description we assume that *checked* is a global variable which may be written or read by all processes. A distributed implementation of *checked* is provided later.

```
Initially: checked = empty, [for all channels: received-marker = false,
           channel-state = pre—marker]

Marker Sending:
           for all channels: channel state = pre-marker →
           send marker along channel; channel state: = positive
```

□

Channel Turning Negative:

for all channels: *channel—state* = *positive and* message is sent along the channel → *channel—state*: = *negative*

□

Setting *Received Marker*:

for all channels: receive a *marker* along the channel → *received-marker*: = *true*

□

Expanding *Checked*:

for all processes *q*: *q* is *unchecked and inc* holds for *q* → *checked*: = *checked* ∪ {*q*}

□

Reinitialization:

for all channels: *channel-state* = *negative* → *channel-state*: = *pre-marker*; *received-marker*: = *false*; **if** the channel is from an *unchecked* process to a *checked* process **then** *checked*: = *empty*

□

Detection:

claim ≡ (*checked* = *P*)

2. Applications of the Paradigm

Now we continue stepwise refinement of the given distributed program. The program outline given above does not specify how the shared variable *checked* is to be implemented. We have several options for refining the program and we describe only one.

We employ one *token* which visits processes one after the other and updates *checked*. When the *token* visits an *unchecked* process the process is added to *checked* if it satisfies *inc*; if the visited process has a *negative* output channel to a *checked* process then *checked* is set to *empty*. How should *negative* channels be reinitialized? The answer is based on the following observation: in the paradigm and its proof we read or write the *channel-state (pre-marker, positive, negative)* and *received-marked* variables only for *unchecked* processes. These variables are not used for *checked* processes. This observation allows us to reset *received-marker* for all input channels of a process and *channel-states* for all its output channels when the process is added to *checked*.

The processes are indexed i, where $0 \leq i < n$, and n is the number of processes in the system. The *token* is implemented as a special message, and for all i, process i sends the *token* to process $(i+1)\, mod\, n$ after completing the computation it is obliged to carry out on receipt of the *token*. The variable *checked* is associated with the *token*; it may be thought of as a field of the *token*. When process i sends the *token* it also sends *markers* on all outgoing channels. Since *marker* and *channel-state* values are reset as the *token* leaves a process, a slight modification of the meanings of these variables is required.

For an input channel of a process:
received-marker is *true* means a *marker* has been received along the channel since the *token* last left the process.

For an output channel of a process:
(*channel-state* = *positive*) means no *black* message has been sent along the channel since the last *marker* was sent along the channel,

(*channel-state* = *negative*) means at least one *black* message was sent along the channel since the last *marker* was sent along the channel.

No channel is in *pre-marker* state.

We now give the algorithm followed by a discussion. The Greek letters $\alpha, \ldots, \epsilon$ are used to label points in the program which are referred to in the discussion. Initial conditions for the algorithm are derived later. The algorithm consists of the *channel turns negative* rule, *setting received-marker* rule and the rule given below.

Algorithm Details For Process i, $0 \leq i < n$

{α::} process i holds the *token and* process i is *white and*
for all its input channels: *received-marker* $\rightarrow$

```
if    process $i$ has a *negative* output channel to a *checked* process
then {$\beta$::} *checked*: = *empty*   else {$\gamma$::} *checked*: = *checked* $\cup$ {$i$};
if    *checked* = $P$ then {$\delta$::} halt
else  {$\epsilon$::} begin
          send *token* to process $(i+1)$ *mod* $n$;
          for all input channels: *received-marker*: = *false*;
          for all output channels: send *marker*; *channel-state*: = *positive*
          end
```
□

When process i satisfies the condition α of the above guarded command then either process i satisfies *inc* (11) or it has a *negative* output channel. In the former case the process is add'd to *checked* {in γ}. In the latter case, if the process has a *negative* output channel to a *checked* process we are obliged to set *checked* to *empty* {in β}. Otherwise (i.e. the process has *negative* output channels only to *unchecked* processes) since the channels will be made *positive* {in ϵ} and since the process will then satisfy *inc* we add it to *checked* {in γ}. Now if *checked* = P the algorithm has detected termination and halts. Otherwise (there is still an *unchecked* process) and so the token is sent the next process and all output channels are made positive {in ϵ}.

The proof of the algorithm is the same as the proof of the paradigm: invariant (12) is maintained, and the same proof of termination applies. (see initial condition below)

2.1. Deriving Initial Conditions for the Algorithm

We derive the initial conditions to ensure progress of the *token* from one process to the next. If we choose the initial conditions unwisely the *token* may get stuck at a process because the condition "for all the process' input channels: *received-marker* holds" may never be met.

Assume that the *token* is initially at process 0 and that all *markers* sent on the "previous" cycle of the token have been received. (Of course, there is no "previous" cycle, but intuition suggests that we determine initial values by assuming that there was). Then, for all channels from higher-numbered processes to lower-numbered processes, *received-marker* holds (because *markers* sent on these channels in the previous cycle are assumed to have been received). For all channels from lower-numbered processes to higher-numbered processes: *not received-marker* (because when the *token* left a process on the last cycle, *received-marker* was set to *false* for all incoming channels, and no marker has since been sent along channels from lower-numbered to higher-numbered processes). Initially, there may be *black* messages in all channels; therefore we assume that all channels are *negative*. Thus we get:

Initially, token is at process 0.

For a channel from a process p to a process q, for all p, q:

received-marker $= (p < q)$.

All channels contain no *markers*.

For all channels: *channel-state* = *negative*.

2.2. Pause To Review Stepwise Refinement

We pause at this point to review the stepwise refinement procedure adopted in this paper. Starting with the problem specification and the notion of set *checked* we showed the need for the inclusion condition and invariant (9). Then we postulated an inclusion condition which resulted in a stronger invariant (12). We next turned our attention to termination and deduced actions to ensure the termination condition in the problem specification. At this point we had the outline for a program though the method of implementation of some global variables (notably *checked*) in a distributed system was left unspecified.

Next, a more complete program outline was obtained by postulating one scheme for implementing the global variables of the previous step so as to maintain the invariant; the implementation used was by means of a *token*. In every step of the refinement we had multiple options and we had to make design choices as to which option to pursue. Different options usually result in different algorithms.

In the next refinement step we show how the program can be optimized by reducing the amount of memory required by each process.

2.3. Implementation Issues

We may reduce the amount of memory required to implement the algorithm by using a few observations about the algorithm.

Notation: Let $i \,..\, j$ denote the set of $(j+n-i)$ *mod* n processes:

i *mod* n, $(i+1)$ *mod* n, ... , j *mod* n.

In other words, $i \,..\, j$ is the set of processes visited by the *token* after i and before next leaving j.

Observation 1: Either *checked* is *empty* or *checked* consists of the last k processes visited by the *token* for some k, where $0 \leq k < n$.

This observation allows us to keep track of the set *checked* by a variable *init* associated with the *token* where *init* has the following meaning. When the *token* arrives at $(j+1)$ *mod* n, if *init* $= j$, then *checked* is *empty* else *checked* is the set $(init+1) \,..\, j$.

Observation 2: The only purpose of *channel-state* is to determine if there is a *negative* channel from an *unchecked* process to a *checked* process.

This observation allows us to implement the algorithm without each process keeping track of *channel-state* for each of its output channels. Each process has a variable *farthest-negative* which is the index of the process "farthest from it" to which it has a *negative* channel, where the sequence of processes ranked in increasing order of "farther from" a process i is:

i, $(i+1)$ *mod* n, $(i+2)$ *mod* n, . . . ,$(i+n-1)$ *mod* n

"Process i has no *negative* output channel," means "*farthest-negative* is i."

Let the *token* be at a process i. "There is a *negative* channel from process i to a *checked* process" is equivalent to, "*farthest-negative* for process i is in the set $init + 1 \,..\, i - 1$". Thus we may dispense with *channel-states* and use a single variable *farthest-negative* for each process. In operational terms, *farthest-negative* for a process is the index of the process farthest from it to which it has sent a *black* message since the *token* last left it.

Observation 3: The variables *received-marker* are used only to determine for a process whether a marker has been received for all its input channels.

This observation allows us to replace variables *received-marker* by a count *nmr* for each process where *nmr* for a process is the *n*umber of *m*arkers *r*eceived by the process since then *token* last left the process. For a process, "*received-markers* holds for all its input channels" is equivalent to "*nmr* = number of input channels of the process".

Algorithm for process i

```
process $i$ holds the token and process $i$ is white and
   nmr = number of its input channels $\rightarrow$
   if    farthest-negative = $i$ and init = $i$
   then {claim = true} halt;
   if    farthest-negative is in the set $(init + 1) \,..\, (i - 1)$
   then init: = $i$;
   send <token, init> to process $(i + 1)$ mod $n$;
   nmr: = 0; send marker on each output channel;
   farthest-negative: = $i$
                                                              □

   receive marker $\rightarrow$ nmr: = nmr + 1
                                                              □
```

```
send *black* message to $j \rightarrow$ if $j$ is farther from $i$ than
  *farthest-negative* then *farthest-negative*: $= j$
```

□

Explanation: For process i, *farthest-negative* $= i$ means the process has no *negative* output channels. If $init = i$ when process i gets the *token* then all processes except i are in *checked*. If *farthest-negative* is in the set $init + 1 \,..\, (i-1)$ then process i has a *negative* channel to a *checked* process and in this case *checked* is set to *empty* and then the *token* is propagated. Setting *checked* to *empty* is accomplished by assigning i to *init* before propagating the *token*.

3. Discussion

Stepwise refinement has been applied in sequential programming to develop programs from specification [11,18]. We have illustrated an application of stepwise refinement to a problem in distributed systems in which the problem specification is in terms of an invariant and termination condition. In distributed systems a useful refinement is that of generalizing predicates on systems to predicates on subsystems, as for instance generalizing W to w. Another useful refinement step is that of implementing global data structures (eg. *checked*) by local data structures and messages (eg. *token*).

Stepwise refinement for the quiescence detection problem yields a family of solutions, one of which was given here. The solution appears to be novel in that attention is restricted to a process's output channels, i.e. *checked* is set to *empty* only if there is a *negative* channel from a *checked* process to an *unchecked* process. Other algorithms use inclusion conditions such as processes being continuously idle (i.e. *white*) over an interval.

It is instructive to study the sequence of system specifications as stepwise refinement proceeds. Invariant (2) and termination condition (3) specify the class of *all* detection problems: a system property W is to be detected, and detection means *claim* holds. In this paper we were concerned with a specific W: quiescence, defined by equation (1). Equation (4) specifies a class of detection algorithms in which detection means "all processes are in *checked*". The choice of w refines the class of solutions further. Every design choice narrowed the set of solutions until we obtained a program. However, the program is less important than the systematic development of design choices because for different environments different solutions are appropriate, and the key question is: Given a development of a program, how much design effort can be saved in developing another program to satisfy a different set of constraints? For example, suppose we did not want a symmetric solution in which all processes are alike (as was given here) but we desired a solution in which one particular process was charged with the responsibility of detection. We can re-use much of the development effort, to solve this problem; indeed we could use the same paradigm. The benefits of systematic development have been discussed for many years, but "calculational" developments of distributed programs are still rare [31]. A point we wish to make by means of the example given here is that it is possible to borrow much of the ideas from sequential program development in writing distributed programs.

We noticed that it was helpful to separate concerns about the development of a concept and its distributed implementation. For example early in the development, we introduced *checked* and w as system-wide variables and later we faced the problem of implementing these variables in a distributed manner on the given processes. Morgan [31] has found it helpful to assume a global clock early in the design process and then later show its distributed implementation. Allowing oneself the latitude of system-wide variables and postponing consideration of distribution appears to be quite helpful. Again, separation of concerns is an idea borrowed from sequential program development; however, the particularly useful manifestation of separation of concerns--postponing issues regarding distribution--does not seems to have received the attention it deserves in the literature on distributed algorithms.

We found that in our development it was helpful to refer to *states* of the program rather than to restrict attention to *observable behaviors* (the messages sent and received by processes). To give a specific instance from our example we proposed the invariant (9):

w(*checked*) *or* for some *unchecked* process: *not inc*

The variable *inc* may or may not be externally observable; while we are developing the algorithm it is helpful to ignore (or at least postpone consideration of) what is externally observable. The danger with referring to states is that solutions may be over-specified; specifications in terms of observable behaviors gives the designer wider latitude in implementation. Though it is sufficient to restrict attention to the sequence of messages sent and received by each process, and though one need not consider a process' local variables, we found it helpful to implement system-wide desiderata in terms of invariants on process' local variables. The tradeoffs between specifying abstract data types in terms of the sequences of observable operations performed on it versus specifying it in terms of (perhaps unobservable) states, have been discussed in the literature on sequential programming. The same tradeoffs are relevant for specifying processes in distributed systems. The point we wish to make here is that in our experience of stepwise refinements of distributed algorithms, we find it helpful to propose invariants in terms of process states. The reason for this is that in the initial stages of refinement we propose invariants on global data structures, and we find it easier to show that these invariants are equivalent to invariants on local data structures than to invariants naming only messages.

This discussion on systematic derivation of distributed algorithms reflects our experience in developing several superimposed algorithms--not merely the single example given here.

Acknowledgements We owe a debt of gratitude to Prof. E. W. Dijkstra for his detailed criticism of earlier drafts. Comments from Hank Korth and Shmuel Katz are appreciated. Discussions at IFIP W.G. 2·3 helped in clarifying our ideas.

References

1. C. Beeri and R. Obermarck, "A Resource Class Independent Deadlock Detection Algorithm", *Research Report RJ3077,* IBM Research Laboratory, San Jose, California, May 1981.

2. G. Bracha and S. Toueg, "A Distributed Algorithm For Generalized Deadlock Detection", *Technical Report TR 83-558,* Cornell University, June 1983.

3. K. M. Chandy and J. Misra, "Asynchronous Distributed Simulation Via a Sequence of Parallel Computations", *Communications of the ACM,* Vol. 24, No. 4, pp. 198-205, April 1981.

4. K. M. Chandy and J. Misra, "A Distributed Algorithm for Detecting Resource Deadlocks in Distributed Systems", *ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing,* Ottawa, Canada, August 1982.

5. K. M. Chandy and J. Misra, "A Computation on Graphs: Shortest Path Algorithms", *Communications of the ACM,* Vol. 25, No. 11, pp. 833-837, November 1982.

6. K. M. Chandy and J. Misra and L. Haas, "Distributed Deadlock Detection", *ACM Transactions on Computing Systems,* Vol. 1, No. 2, pp. 144-156, May 1983.

7. K. M. Chandy and L. Lamport, "Distributed Snapshots: Determining Global States of Distributed Systems", to appear in *ACM Transactions on Computing Systems.*

8. E. Chang, "Echo Algorithms: Depth Parallel Operations on General Graphs", *IEEE Transactions on Software Engineering,* Vol. SE-8, No. 4, pp. 391-401, July 1982.

9. S. Cohen and D. Lehmann, "Dynamic Systems and Their Distributed Termination", *ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing,* pp. 29-33, Ottawa, Canada, August 18-20, 1982.

10. E. W. Dijkstra and C. S. Scholten, "Termination Detection for Diffusing Computations", *Information Processing Letters,* Vol. 11, No. 1, August 1980.

11. E. W. Dijkstra, "A Discipline of Programming", Prentice Hall, 1976.

12. E. W. Dijkstra, "Distributed Termination Detection Revisited", EWD 828, Plataanstraat 5, 5671 AL Nuenen, The Netherlands.

13. N. Francez, "Distributed Termination", *ACM Transactions on Programming Languages and Systems,* Vol. 2, No. 1, pp. 42-55, January 1980.

14. N. Francez, M. Rodeh, and M. Sintzoff, "Distributed Termination with Interval Assertions", *Proceedings of Formalization of Programming Concepts,* Peninsula, Spain, April 1981. Lecture Notes in Computer Science 107, (Springer-Verlag).

15. N. Francez and M. Rodeh, "Achieving Distributed Termination Without Freezing", *IEEE-TSE,* Vol. SE-8, No. 3, pp. 287-292, May 1982.

16. V. Gligor and S. Shattuck, "On Deadlock Detection in Distributed Data Bases", *IEEE Transactions on Software Engineering,* Vol. SE-6, No. 5, September 1980.

17. M. Gouda, "Personal Communication", Department of Computer Sciences, University of Texas, Austin, Texas 78712.

18. D. Gries, "The Science of Programming", Springer-Verlag, 1981.

19. L. Haas and C. Mohan, "A Distributed Deadlock Detection Algorithm for a Resource-Based System", *Research Report RJ3765,* IBM Research Laboratory, San Jose, California, January 1983.

20. T. Herman and K. M. Chandy, "A Distributed Procedure to Detect AND/OR Deadlock", Computer Sciences Department, University of Texas, Austin, Texas 78712, February 1983.

21. T. Holt, "Some Deadlock Properties of Computer Systems", *Computing Surveys,* Vol. 4, No. 3, pp. 179-196, September 1972.

22. D. Kumar, Ph.D Theses (in preparation), Computer Sciences Department, University of Texas, Austin, Texas 78712.

23. L. Lamport, "Time, Clocks and the Ordering of Events in a Distributed System", *Communications of the ACM,* Vol. 21, No. 7, July 1978.

24. G. Le Lann, "Distributed Systems - Towards a Formal Approach", *Information Processing 77,* IFIP, North-Holland Publishing Company,

1977.

25. D. Menasce and R. Muntz, "Locking and Deadlock Detection in Distributed Data Bases", *IEEE Transactions on Software Engineering,* Vol. SE-5, No. 3, May 1979.

26. J. Misra and K. M. Chandy, "A Distributed Graph Algorithm: Knot Detection", *ACM Transactions on Programming Languages and Systems,* Vol. 4, No. 4, pp. 678-688, October 1982.

27. J. Misra and K. M. Chandy, "Termination Detection of Diffusing Computations in Communicating Sequential Processes", *ACM Transactions on Programming Languages and Systems,* Vol. 4, No. 1, pp. 37-43, January 1982.

28. J. Misra, "Detecting Termination of Distributed Computations Using Markers", *Proceedings of the ACM SIGACT-SIGOPS Symposium of Principles of Distributed Computing,* Montreal, Canada, August 17 - 19, 1983.

29. R. Obermarck, "Deadlock Detection For All Resource Classes", *Research Report RJ2955,* IBM Research Laboratory, San Jose, California, October 1980.

30. R. Obermarck, "Distributed Deadlock Detection Algorithm", *ACM Transactions on Database Systems,* Vol. 7, No. 2, pp. 187-208, June 1982.

31. Dijkstra, E.W., Feijen, W.H.J., & Van Gasteren, A.J.M., "Derivation of a termination detection algorithm for distributed computations" *Information Processing Letters*, Vol. 16, pp. 217-219, 1983.

32. Morgan, Caroll, "Relaxing Distributed Algorithms" to appear *Information Processing Letters*, 1984.

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER	2. GOVT ACCESSION NO.	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) Manuscript: "A Class of Termination Detection Algorithms for Distributed Computations"		5. TYPE OF REPORT & PERIOD COVERED final: 6/14/81 - 6/15/85
		6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s) Devendra Kumar (Graduate Student) University of Texas at Austin		8. CONTRACT OR GRANT NUMBER(s) AFOSR 81-0205
9. PERFORMING ORGANIZATION NAME AND ADDRESS Computer Sciences Department University of Texas at Austin Austin, Texas 78712		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
11. CONTROLLING OFFICE NAME AND ADDRESS Capt. A. L. Bellamy AFOSR/NM Bolling AFB, DC 20332		12. REPORT DATE July 1985
		13. NUMBER OF PAGES
14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office)		15. SECURITY CLASS. (of this report)
		15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

submitted to IEEE Transactions on Software
(Technical Report TR-85-07, Computer Sciences Department, UT)

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

We present a class of efficient algorithms for termination detection in a distributed system. These algorithms do not require the FIFO property for the communication channels. Assumptions regarding the connectivity of the processes are simple. Messages for termination detection are processed and sent out from a process only when it is idle. Thus it is expected that these messages would not interfere much with the underlying computation, i.e., the computation not related to termination detection. The messages have a fixed, short length.

After termination has occurred, it is detected within a small number of message communications.

The algorithms use markers for termination detection. By varying assumptions regarding connectivity of the processes, and the number of markers used, a spectrum of algorithms can be derived, changing their character from a distributed one ot a centralized one. The number of message communications required to detect termination after its occurrence depends on the particular algorithm — under reasonable connectivity assumptions it varies from order N (where N is the number of processes) to a constant.

This paper introduces message counting as a novel and effective technique in designing termination detection algorithms. The algorithms are incrementally derived, i.e., a succe-sion of algorithms are presented leading to the final algorithms. Proofs of correctness are presented. We compare our algorithms with other work on termination detection.

A CLASS OF TERMINATION DETECTION ALGORITHMS FOR DISTRIBUTED COMPUTATIONS[1]

Devendra Kumar

Department of Computer Sciences
University of Texas at Austin
Austin, Texas, 78712

TR-85-07 May 1985

[1]This work was supported by Air Force Grant AFOSR 81-0205.

Abstract

We present a class of efficient algorithms for termination detection in a distributed system. These algorithms do not require the FIFO property for the communication channels. Assumptions regarding the connectivity of the processes are simple. Messages for termination detection are processed and sent out from a process only when it is idle. Thus it is expected that these messages would not interfere much with the underlying computation, i.e., the computation not related to termination detection. The messages have a fixed, short length. After termination has occurred, it is detected within a small number of message communications.

The algorithms use markers for termination detection. By varying assumptions regarding connectivity of the processes, and the number of markers used, a spectrum of algorithms can be derived, changing their character from a distributed one to a centralized one. The number of message communications required to detect termination after its occurrence depends on the particular algorithm – under reasonable connectivity assumptions it varies from order N (where N is the number of processes) to a constant.

This paper introduces message counting as a novel and effective technique in designing termination detection algorithms. The algorithms are incrementally derived, i.e., a succession of algorithms are presented leading to the final algorithms. Proofs of correctness are presented. We compare our algorithms with other work on termination detection.

1. Introduction

We develop a class of efficient algorithms for termination detection in a distributed system. We do not require the FIFO property for the communication channels, which is usually assumed in other works. (The FIFO property for a communication channel means that messages in the channel are received in the same order as they were sent.) Our assumptions regarding connectivity of processes are simple. We have categorized our algorithms in three classes. Algorithms in classes 1 and 2 assume that there exists a cycle involving all processes in the network. This cycle need not be an elementary cycle, i.e., a process may be arrived at several times in a traversal of the cycle. Moreover, the edges of the cycle need not be *primary edges*, i.e., the edges involved in the underlying computation; *secondary edges* may be introduced in the network to facilitate termination detection. (We use the terms *edges*, *lines*, and *channels* interchangeably.) Normally the length of this cycle would affect performance of the algorithms; by using secondary edges, if necessary, the length of this cycle may be kept to a minimum. Algorithms in class 3 assume the existence of cycles in several parts of the networks.

In these algorithms, messages for termination detection are processed and sent out from a process only when it is idle. Thus it is expected that these messages would not interfere much with the underlying network computation, i.e., the computation whose termination is to be detected.

Except for algorithms in class 1, the messages for termination detection in these algorithms have a fixed, short length (a pair of integers). In all algorithms presented, termination is detected within a small number of message communications after its occurrence.

In devising an algorithm for detecting termination, deadlock, or some other stable property [Chandy 85a, Chandy 85b], one important issue is how to determine if there are no primary messages in transit (*primary messages* are those transmitted in the underlying computation; *secondary messages* are those related to termination detection). Several approaches have been developed to handle this issue – acknowledgement messages [Chandy 85a], using a marker to "flush out" any messages in transit (with the assumption of FIFO property) [Misra 83, Chandy 85a], etc. One contribution of this paper is to suggest a new approach – counting the number of primary messages sent and received. As shown in this paper, this approach has several desirable features – it results in simple and flexible connectivity requirements, it does not require the FIFO property for the communication channels, and it does not generate too much overhead in terms of the number of secondary messages after the occurrence of termination. Moreover, we show that it is not necessary to count and transmit information regarding number of primary messages on *individual lines* – it is sufficient to count and transmit information about the total number of primary messages received and the total number of primary messages sent by individual processes.

Classification of Our Algorithms

Algorithms in class 1 are based on counting primary messages on every line. Each process keeps a count of the number of primary messages it has received or sent on each adjacent line (i.e., input line or output line respectively). As mentioned above, algorithms in class 1 assume that there exists a cycle C including every process of the network at least once. A marker traverses the cycle, and uses these counts in detecting termination. After termination has occurred, it will be detected within |C|-1 communications of the marker. (|C| refers to the length of the cycle C, i.e., the number of edge traversals required to complete the cycle.) The problem with this algorithm is that each message is long – it consists of E number of integers where E is the total number of primary lines in the network.

Algorithms in class 2 reduce the message length. In these algorithms, each process counts the total number of primary messages received by it, and the total number of primary messages sent by it. Here counts are not being kept for individual adjacent lines. A marker traverses the cycle C, and collects this information to detect termination. In this case the message length is short (two integers). After the occurrence of termination, it will be detected within $2 \cdot |C| - 2$ message communications. Note that if C is an elementary cycle then $|C| = N$, where N is the number of processes in the distributed system.

Next, class 3 of our algorithms improve the performance of the algorithms in class 2, by using multiple markers which traverse different parts of the system. We make simple connectivity assumptions to permit these traversals. Using two markers, under reasonable assumptions the number of message communications after the occurrence of termination is reduced to approximately 3N/2, each message carrying an integer and a boolean. As the number of markers is increased, this number reduces further and the algorithm tends to change its character to a centralized one. Finally, using N markers, this number is reduced to the constant 4, and the algorithm becomes a purely centralized one.

On the Nature of This Presentation

A number of excellent papers on deadlock and termination detection for distributed systems have appeared in recent years. These papers usually discuss how the algorithm executes, i.e, what are the key data or execution steps in the algorithms. Proofs of correctness are usually provided to convince the reader that the given algorithm works. However, certain other important questions are usually left unanswered. How was the algorithm developed in the first place? Why were certain decisions (conventions, assumptions, major data, major execution steps) in the design of the algorithm taken – are they critical to the correctness, or are they present simply to enhance performance, or understandability, etc.? How would a simple variation of these decisions affect either correctness or performance? For the algorithms discussed here, our presentation attempts to answer some of these questions to a certain extent. We discuss a succession of algorithms, each algorithm differing from the previous ones in a simple manner. Several simple variations of the algorithms are considered. As would be noted in the discussion, some of these "algorithms" are not even correct; they are discussed simply to enhance understandability of later algorithms. Moreover, specific details, for instance initial conditions, are derived from more general considerations. It is hoped that with this method of presentation, the reader can develop a better insight as to how various decisions were arrived at. Since the relationships among various algorithms are explicitly discussed, this approach would also help keep a clear and organized view of the class of algorithms presented.

Related Work

Termination detection in distributed systems has been a subject of much study in recent years. One of the earliest works in this area is the elegant algorithm of [Dijkstra 80]. This is one of the few algorithms that do not assume the FIFO property for the communication channels. However, this algorithm requires that for any primary line from a process i to a process j, there must be a line from j to i. Termination is detected within N message communications after its occurrence, where N is the number of processes in the system. However, depending on the nature of the underlying computation, *in the entire computation* the total number of secondary messages generated in this algorithm may be too much. (The total number of secondary messages in this algorithm is equal to the total number of primary messages.) This may severely affect performance. Moreover, secondary messages are processed and sent out from a process even when it is active. This may slow down the underlying computation itself.

The above algorithm was extended in [Misra 82a] to CSP [Hoare 78] environment. The basic idea of the algorithm has been used in several distributed algorithms in many application areas [Cohen 82, Misra 82b, Chandy 82b, Chandy 81].

Marker based algorithms usually do not suffer from the drawbacks mentioned above for the algorithm in [Dijkstra 80]. A marker is sent from a process only when it is idle. Therefore normally the secondary computation would not significantly slow down the underlying computation. (*Secondary computation* is that related to termination detection; the underlying computation is also called the *primary computation*.) Moreover, usually the total number of secondary messages would also be small. Roughly speaking, if the primary computation becomes more intense (i.e., primary messages are being generated at a higher rate), then the recipient processes are likely to be active more of the time (i.e., idle for lesser time). Hence the marker is likely to move less frequently since it has to wait till the process has become idle. However, this is not to say that marker based algorithms always result in better performance; in fact many such algorithms require more than N message communications after the occurrence of termination.

Distributed termination detection using a marker was devised by Francez et. al. [Francez 80, Francez 81, Francez 82]. This approach was improved upon, removing some of its restrictions, in another marker based algorithm [Misra 83]. In this algorithm a marker traverses a cycle C' that includes every edge of the network at least once. The algorithm requires the FIFO property for the communication channels. Termination is detected, after its occurrence, within two rounds of this cycle. Note that, in principle, assuming the existence of a cycle traversing every edge is equivalent to assuming the existence of a cycle traversing every process (as in our approach). However, the performance resulting from the two approaches would normally be different. The cycle C' in general may be quite large – usually it would be longer than the total number of primary edges in the network, and the number of primary edges can be $O(N^2)$. In contrast, in our approach we can always define an elementary cycle (whose length will be N), introducing secondary edges if necessary. Defining an optimal or near optimal cycle in our approach is much simpler, since we don't require the cycle to involve every primary edge. If the network is evolving over time (e.g., new primary lines or processes being added to the network) our approach would normally require simpler changes in the data stored at the processes regarding this cycle.

In several recent works [Chandy 85a, Chandy 85b) the notion of termination and deadlock has been generalized and elegant schemes have been presented to solve these general problems. [Chandy 85a] shows how the general scheme presented there can be applied in many ways to solve the specific problems of termination and deadlock detection. The termination detection algorithm described there assumes the FIFO property for the communication channels. A marker traverses a cycle that includes every process of the network at least once. Termination is detected, after its occurrence, within two rounds of this cycle. The marker is a short message, containing only one integer. However, before the marker is sent out from a process, another message (containing no data) is sent out on output lines of this process. This effectively doubles the number of message communications after occurrence of termination. Since our algorithms in class 2 involve two rounds of the same cycle, with each secondary message having two integers, we expect comparable performance between the above algorithm and our algorithms in class 2. However, our schemes in class 3 improve the performance even further. As indicated in [Chandy 85a], the FIFO requirement for the communication channels may be removed, leading to another algorithm. But that algorithm would involve too many acknowledgement messages (equal to the number of primary messages).

One nice property that the two algorithms above ([Misra 83] and the algorithm in [Chandy 85a] using the FIFO property) enjoy is that the termination detection algorithm may be initiated with the underlying computation of the network in an arbitrary state, i.e., there may be an arbitrary number of primary messages in transit and the processes may be in arbitrary states. Our algorithms and most of the other algorithms published require special initializations for the secondary computation before the underlying computation starts.

As mentioned earlier, termination detection has been used in designing several other distributed algorithms. Many distributed algorithms can be devised as multiphase algorithms, where a new phase is started after the termination detection of the previous phase. Distributed simulation schemes have been devised using this approach [Chandy 81, Kumar 85]. [Francez 81] suggests a methodology for devising distributed programs using termination detection.

A problem of considerable importance that is closely related to termination detection, is the problem of deadlock detection in distributed systems. Several important pieces of works have appeared in this area [Gligor 80, Beeri 81, Obermarck 82, Chandy 82a, Chandy 83, Bracha 83, Haas 83].

Synopsis of the Rest of the Paper

Section 2 defines the model of computation and defines the termination detection problem. Criteria used for comparing termination detection algorithms may vary widely – performance, storage requirements, communication cost, simplicity of implementation, etc. In this paper we concern ourselves only with performance. In section 3 we discuss our performance criteria. Sections 4, 5, and 6 discuss our algorithms in classes 1, 2, and 3 respectively (we have commented on these classes earlier in the introduction). Finally section 7 gives concluding remarks.

2. Problem Definition

First we describe a basic model of a distributed system. For ease of exposition, we discuss our algorithms in terms of this basic model. Our algorithms are applicable to more general distributed systems; we briefly mention these systems later in this section.

The Basic Model

A distributed system consists of a finite set of processes, and a set of unidirectional *communication channels* (or *lines*, or *edges*). Each communication channel connects two distinct processes. Given two processes i and j, there is at most one communication channel from i to j, denoted by the ordered pair (i, j).

In addition to their local computations, processes may send or receive messages. Process i can send a message to process j only if the line (i, j) exists. Process i does so by depositing the message in the channel (i, j). This message arrives at process j after an arbitrary but finite (possibly zero) delay. Process j receives the message by removing it from the channel, within a finite time (possibly zero) of its arrival. The channels are error-free, except that they need not be FIFO channels.

The "Underlying" Computation

Now we describe the nature of the computation (called the *underlying computation* or the *primary computation*) whose termination is to be detected. The messages sent or received in this computation are called *primary messages*. Later other computation (called *secondary computation*) would be superimposed on this computation for the purpose of termination detection.

From the point of view of the underlying computation, at any moment a process is in one of two states:

1. *Active state:* In this state, a process may send primary messages on its outgoing lines. It may become idle at any time.

2. *Idle state:* In this state, a process can not send any primary messages. On receiving a primary message, it may remain idle or switch its state to active.

A process in any of the two states may receive primary messages or do any local computations. It is assumed that initially, (i.e., when the primary computation starts) there are no primary messages in transit; though the processes may be in arbitrary states.

The Termination Detection Problem

A message in the distributed system is said to be a *transient message* if it has been sent, but has not been received yet. We say that at a moment t the distributed computation is terminated iff:

1. all the processes are idle at time t, and

2. there are no transient primary messages at time t.

It is obvious that if the network computation is terminated at a time instant t, then it would remain terminated for all times after t (unless forced otherwise by some outside agent). The problem is to detect the state of termination within a finite time after its occurrence. To this end, we will devise an algorithm to be superimposed on the underlying computation; this algorithm must satisfy the following properties:

1. Termination is reported, to some process in the network, within a finite time after termination of the underlying computation, and

2. if termination is reported at some time t, the network must be terminated at time t (i.e., no "false detection" of termination is allowed).

Messages related to termination detection are called *secondary messages.* It may be noted that an idle process may send secondary messages, even though it can not send primary ones.

Other Models

We briefly mention here other features that could be incorporated in our model of computation, without affecting the applicability of our algorithms (possibly with some minor modifications).

1. We may allow multiple communication channels from a process i to a process j. Also, a process could be allowed to send a message to itself. These extensions may be useful if a process consists of a set of interacting subprocesses.

2. A process may *broadcast* a message to a set of processes. This is equivalent to sending the same message via communication lines to each process in the set.

3. There may be a third state for a process — a *terminated state.* A process enters this state when it is guaranteed that it will not send out any primary messages in future, and no more primary messages would arrive at its input lines.

3. Performance Criteria

There are two major criteria for performance evaluation of termination detection algorithms:

1. The effect of secondary computation on the primary computation itself, i.e., how the primary computation gets slowed down and

2. How long it takes to detect termination after its occurrence.

In general, the two criteria above would be assigned different weights, depending on the objectives of the primary computation and its termination detection. One has to consider not only the time delays involved, but also how *time critical* the two delays are. Depending on application, one of these may carry a higher weight than the other. The following examples illustrate this:

1. Consider a distributed system that monitors a physical system. The primary computation is triggered by an extreme state in the physical system and its objective is to bring the system to a steady state. The primary computation terminates after the system returns to the steady state. Here the former criterion would be more significant.

2. Consider a secondary computation whose objective is to detect the termination of a token in a token ring [Misra 83]. Suppose the loss of the token represents an extreme state that must be corrected immediately. Here the second criterion would be more significant.

3. Consider a multiphase distributed simulation [Chandy 81, Kumar 85]. Here

the objective is to reduce the *total* simulation time. In this case none of the two delays above are time critical and both affect the overall objective in the same way; thus both criteria would have equal weights here.

In this paper we will focus on the second criterion. (As mentioned above, in a particular application this may or may not be a good criterion for performance measurement.) Let I denote the time interval between the occurrence of termination and its detection.

How should one estimate I? Obviously, the value of I depends on characteristics of the system that supports the primary computation. We use the number of (secondary) message communications during the interval I and the lengths of these messages as a measure of I. Knowing the characteristics of communication delays, one may establish either I or an upper bound on it. For simplicity of discussion, we assume that any communication delay in the system is a linear function of message length. We mention below a few details about our performance evaluation:

1. Note that the value of I (and the associated measures mentioned above) would depend on where the marker is at the time when termination occurs, etc. For simplicity, we would normally consider only the worst case values.

2. Message communications at the same time on different lines will be taking place *in parallel* – this must be taken into account in determining the number of messages, i.e., during any overlapping period, only one message is considered being communicated. In general, any two independent events will be assumed to take place in parallel.

3. During the interval I, the number of messages received may be different (slightly) from the number of messages sent. We consider the latter one as the number of message communications. (This would be more reasonable in situations where the time involved in the act of sending a message, i.e., the *transmission time*, is longer than the propagation delay of the message.)

4. By message length we mean the total length of data in it. It is assumed that even for a message of length zero, there would be a non-zero communication delay.

4. Class 1 of Algorithms: Counting Primary Messages on Each Line

In these algorithms, a marker traverses a cycle C that includes every process of the network at least once (discussed in section 1). Information as to how many messages are in transit is kept by counting the number of primary messages sent, and received, for each line. Each process i has two local arrays $SNTP_i$ and $RECP_i$. (For simplicity of discussion, we assume here that primary lines in the network are globally numbered 1, 2, ..., E and each array $SNTP_i$ and $RECP_i$ has E elements. We will discuss more appropriate data structures later.) At any time, $SNTP_i(e)$ = the number of primary messages sent by process i on line e after the last visit of marker at i (or since the initial time, if the marker has not visited i yet). $RECP_i(e)$ is similarly defined for messages received. Each process i increments $SNTP_i(e)$ or $RECP_i(e)$, respectively, on sending or receiving a primary message on line e.

The marker has two arrays *SNTM* and *RECM*, where it keeps its knowledge as to how many primary messages have been sent or received on each line.

(For convenience, in this paper we use the obvious notation for *array assignments*, *array equality*, etc. Also, we often use a time argument in a variable to refer to its value at that time.)

An Algorithm-Skeleton

The following basic algorithm-skeleton is followed by the marker.

```
(* marker arrives at process i, i.e. it is received by i. *)
    The marker waits till process i becomes idle;

(* Process i is idle now. Marker starts its visit at i. *)
    $SNTM := SNTM + SNTP_i$;
    $SNTP_i := 0$;
    $RECM := RECM + RECP_i$;
    $RECP_i := 0$;
(* The visit at process i is completed. *)

(* Declare termination or depart from process i. *)
    Under an appropriate condition (to be discussed) the marker
    declares termination. If this condition does not hold,
```

the marker leaves process i along the next line on cycle C.

We discuss later (under the heading "some improvements and details") the algorithm and data structures required to facilitate the repeated traversal of the cycle C by the marker.

A process does not receive any messages during the interval between the start of marker's visit and its departure. In other words, the underlying computation at a process is carried out only before the marker's visit and after its departure. As mentioned earlier, the variables $SNTP_i$ and $RECP_i$ are incremented on sending or receiving (respectively) a primary message.

The variables related to termination detection are initialized before the primary computation starts. Initially, a value of zero is assigned to all elements of *SNTM*, *RECM*, $SNTP_i$, and $RECP_i$. (This initialization will be changed later in the discussion.) Also, the marker is initially at an arbitrary process, and visits it when the process becomes idle.

The above is only a skeleton of an algorithm; we have not yet discussed when the marker declares termination. We address this issue now. Suppose the primary computation terminates at time T_f. Then within a finite time after T_f the system would reach a state where the condition $SNTM = RECM$ is true (i.e., the corresponding elements of the two arrays are equal) and would remain true forever. (After T_f this condition may become true or false several times, but definitely after one complete traversal of the cycle C by the marker it will remain true forever.) This is stated as theorem 1 below. This suggests a way of detecting termination, but we still have to avoid the possibility of detecting "false termination". Note that the condition $SNTM = RECM$ being true at a point in computation does not guarantee that termination has occurred. For example, initially this condition holds, but the system may have active processes. We ask the question – suppose in a sequence of visits along the cycle C, the marker continuously finds that $SNTM = RECM$. Can it conclude termination after a (predefined) finite number V of such visits? Theorem 2 looks at this question in a 'brute force' manner, and answers it in the affirmative with $V = 2.|C|$. Using this theorem one can complete the algorithm. Thereafter we consider the question of efficiency. Theorem 3 improves the efficiency of this algorithm by reducing V to $|C|$. Theorem 4 provides a way of reducing V to 1 if an additional condition is guaranteed before announcing termination. Later we discuss how to ensure this condition in an efficient way. (It will be observed that as we progress from theorem 2 towards theorem 4, the results become less obvious and the proofs of correctness more complex.) Let us first discuss some intermediate results that will be used in the proofs of these theorems.

For convenience, in this paper we will be implicitly using the convention that events are totally ordered, e.g., as in [Misra 81]. The events of interest are – sending a primary message, receiving a primary message, a process changing its state, and the marker arriving at a process, starting a visit, completing a visit, and departing the process. All time instants mentioned in this paper correspond to a point in the trace of events in the system, unless otherwise specified. In particular, normally no time instant refers to a moment in between the start and completion of a visit. (Otherwise many of our lemmas will become incorrect!)

Let *tsnt*(e, t) = the total number of messages sent on line e up to (including) time t.

trec(e, t) is similarly defined for messages received.

Let r(e, t) = the number of transient messages on line e at time t.

Lemma 1: For any line e and any time t:

$$tsnt(e, t) = trec(e, t) + r(e, t) \quad (1)$$

and $$tsnt(e, t) \geq trec(e, t) \quad (2)$$

Proof: Follows from the definitions.

Lemma 2: For any line e and any two time instants t, t' such that t < t':

$$tsnt(e, t) \leq tsnt(e, t') \quad (3)$$

and $trec(e, t) \leq trec(e, t')$ (4)

Proof: Follows from the definitions.

Lemma 3: For any line e = (i, j) and for any time t:

$$tsnt(e, t) = SNTM(e, t) + SNTP_i(e, t) \tag{5}$$

and $trec(e, t) = RECM(e, t) + RECP_j(e, t)$ (6)

Proof: The proof is by induction on the number of events in the system [Misra 81]. Initially, (5) and (6) are true. Also, each event leaves any of them invariant.

Lemma 4: For any line e = (i, j) and for any time t:

$$r(e, t) = SNTM(e, t) - RECM(e, t) + SNTP_i(e, t) - RECP_j(e, t) \tag{7}$$

Proof: Follows from (1), (5), and (6).

Lemma 5: Consider a "current" moment T in computation. For a line e = (i, j), suppose both processes i and j have been visited by the marker at least once. Let t_i and t_j, respectively, be the last times at which visits at processes i and j were completed. Then,

$$SNTM(e, T) - RECM(e, T) = tsnt(e, t_i) - trec(e, t_j) \tag{8}$$

Proof: Obviously $SNTM(e, T) = SNTM(e, t_i)$, $RECM(e, T) = RECM(e, t_j)$, $SNTP_i(e, t_i) = 0$, and $RECP_j(e, t_j) = 0$. The result follows from lemma 3.

Note: Later we will make certain changes that will make lemma 2 incorrect. However, lemmas 1, 3-5 will not be affected. Proofs of theorems 1-6 below will rest only on lemmas 1, 3-5 – they will not use lemma 2 directly.

Theorem 1: If the underlying computation terminates at a time T_f, then within a finite time after T_f the system would reach a state where the condition *SNTM* = *RECM* is true and would remain true forever thereafter (until termination is declared and possibly a new primary computation is started).

Proof: After T_f, all processes remain idle forever; therefore the marker does not wait indefinitely after its arrival at a process. Hence, within a finite time after T_f (say at a time T, $T \geq T_f$), the marker would have made a complete traversal of the cycle C, i.e., it would have visited every process at least once after time T_f (unless it has declared termination earlier). Let $T' \geq T$ be any "current" time. For any line e = (i, j) let t_i and t_j, respectively, be the last times at which processes i and j were visited. Obviously, $t_i \geq T_f$ and $t_j \geq T_f$. From lemma 5,

$$SNTM(e, T') - RECM(e, T') = tsnt(e, t_i) - trec(e, t_j)$$

But $tsnt(e, t_i) = tsnt(e, T_f)$, $trec(e, t_j) = trec(e, T_f)$, and $tsnt(e, T_f) = trec(e, T_f)$. The result follows.

$\blacksquare$

Theorem 2: Suppose in a sequence of V = 2.|C| visits, the marker continuously finds the condition *SNTM* = *RECM* to be true after each visit in the sequence. Then at the end of this sequence it can conclude that the underlying computation has terminated.

Proof: Let T_0 be the time when the marker has completed |C| number of visits in the above sequence. We will show that at time T_0 the primary computation is terminated. Let t_{i0} be the time at which the marker completed its last visit at process i up to (including) time T_0. Also, let t_{i1} and t_i be the times at which the marker started and finished, respectively, its fist visit at process i after time T_0 (here we are considering the start and completion of a visit as two distinct events in the history of events in the system). Obviously, for all i $t_{i0} \leq T_0 < t_{i1} < t_i$.

We first show that at time T_0, for all primary lines e = (i, j), $SNTP_i(e, T_0) = 0$. In a similar manner it can be shown that $RECP_j(e, T_0) = 0$. Suppose for some e = (i, j), $SNTP_i(e, T_0) > 0$. Obviously $SNTP_i(e, t_{i1}) \geq SNTP_i(e, T_0) > 0$. Hence $SNTM(e, t_i) = SNTM(e, t_{i1}) + SNTP_i(e, t_{i1}) > SNTM(e, t_{i1})$. But $SNTM(e, t_{i1}) = RECM(e, t_{i1})$ and $RECM(e, t_{i1}) = RECM(e, t_i)$. Therefore $SNTM(e, t_i) > RECM(e, t_i)$. This contradicts the hypothesis of the theorem.

Since for every line e = (i, j), $SNTP_i(e, T_0) = RECP_j(e, T_0) = 0$ and $SNTM(e, T_0) = RECM(e, T_0)$, it follows from lemma 4 that $r(e, T_0) = 0$. In other words there are no transient primary messages at time T_0.

Now we show that every process i is idle at time T_0. Obviously i is idle at time t_{i0}. Also, i did not receive any primary messages during the interval $[t_{i0}, T_0]$, otherwise we will have $RECP_i(e, T_0) > 0$ for the corresponding input line e, which will contradict the above result that $RECP_i(e, T_0) = 0$. Thus i is idle at time T_0. This completes the proof.

$\blacksquare$

Theorem 3: Theorem 2 remains valid if the requirement $V = 2.|C|$ in it is changed to $V = |C|$.

Proof: Let T_0 and T, respectively, be the times when the first and the last visits in the sequence are completed. For any process i, choose any particular visit that was completed in the interval $[T_0, T]$ and let t_{i1} and t_i, respectively, be the times at which this visit was started and finished. Claim (A) below can be shown easily (if i is the first process visited in the sequence and $t_i = T_0$, then (A) follows readily; for other cases it follows as in the proof of theorem 2):

(A) At any time t during the interval $[T_0, t_i]$, process i has $SNTP_i(e, t) = RECP_i(e, t) = 0$ for any adjacent primary line e.

Since $SNTM = RECM$ after each visit in the sequence, from (A) we conclude that:

(B) At time T_0 there are no transient primary messages, and

(C) Process i did not send or receive a primary message in the interval $[T_0, t_i]$.

Note that a process i may be active at time T_0. We will show that after time t_i, process i will never receive a primary message. Since any message in transit will be received after a finite time, this proves that there are no transient messages at time T when the above sequence of visits is completed. Moreover, since process i is idle at time t_i and does not receive any primary messages after time t_i, it will be idle at time T.

We say that a primary message is a *bad* message if it is received at a process i after time t_i. We will prove by contradiction that there can be no bad messages in the system. Suppose there are bad messages in the system. Let m be the bad message with the earliest time of reception (say t_r). Suppose m was sent on a line $e = (i, j)$ at time t_s. Obviously, $t_r > t_s$ and $t_r > t_j$. Consider the following two cases.

Case 1: $t_s > t_i$, i.e., m was sent out after the marker's last visit at i. Then process i must have received a bad message after t_i and before t_s (hence before t_r). this contradicts the assumption that m is the bad message with the earliest time of reception.

Case 2: $t_s < t_i$. We have shown above (C) that process i does not send any primary messages in the interval $[T_0, t_i]$. Therefore m must have been sent before T_0. Hence m is in transit at time T_0. This contradicts (B) above. This completes the proof.

$\blacksquare$

Now we attempt to reduce further the length of the sequence of visits required with the condition *SNTM* = *RECM* before the marker can conclude termination. Note that in order to detect termination, the marker must visit every process at least once after the start of the secondary computation; since in our scheme the state (idle or active) of a process can not be deduced from the information available at the other processes. We show below in theorem 4 that if every process has been visited at least once, then the condition *SNTM* = *RECM* after visiting a process guarantees that termination has indeed occurred.

Theorem 4: Suppose, after visiting a process, the marker finds that *SNTM* = *RECM*.

Also, suppose the marker has visited every process at least once by this time. Then at this time T the underlying computation is in the terminated state.

Proof: Let t_i be the last time that the marker completed its visit at process i up to time T (i.e., $t_i \leq T$). We will show that after time t_i, process i would never receive a primary message. As argued in the proof of theorem 3, this leads to the conclusion.

With the above definition of t_i, we define *bad messages* in the same way as in the proof of theorem 3. The argument continues as before and case 1 is the same. Case 2 is different now and we consider it below.

Case 2: $t_s < t_i$, i.e., m was sent before the marker last visited process i. Since *SNTM*(e, T) = *RECM*(e, T), from lemma 5 we get *tsnt*(e, t_i) = *trec*(e, t_j). Consider the following two subcases.

Case 2.1: $t_i < t_j$. By definition of m, process i did not receive any primary messages in the interval $[t_i, t_j]$. Therefore process i did not send any primary messages in this interval. Therefore, *tsnt*(e, t_i) = *tsnt*(e, t_j). Hence *tsnt*(e, t_j) = *trec*(e, t_j). But there is at least one transient message, namely m, on line e at time t_j (since m was sent before t_i and received after t_j). This contradicts (1).

Case 2.2: $t_j < t_i$. Since *tsnt*(e, t_i) = *trec*(e, t_j), using (2) and (3) we conclude in this case that *tsnt*(e, t_i) = *tsnt*(e, t_j) = *trec*(e, t_j). In other words, no primary messages were sent on line e during $[t_j, t_i]$ and there are no transient messages on line e at time t_j. Hence m was sent before t_j and received by the time t_j. This contradicts with the definition of m.

$\blacksquare$

Note: The proof of case 2 will be simpler if one assumes the FIFO property for the communication channels. Informally, since m has been counted in *tsnt*(e, t_i) and has not been counted in *trec*(e, t_j), by the FIFO property we will get *tsnt*(e, t_i) > *trec*(e, t_j). Therefore we won't have to consider the cases 2.1 and 2.2.

Completion of the Algorithm

It may be noted that if the hypothesis of theorem 2 or theorem 3 is true then the hypothesis of theorem 4 is true as well, but not vice versa. Therefore the method suggested by theorem 4 would be more efficient. Hence we use theorem 4 to complete the algorithm. How would the marker decide that it has visited every process at least once? One brute force method would be to have a counter in the marker that counts how many visits have been completed. When this counter becomes $|C|$, obviously every process has been visited at least once. (Alternatively, the marker could count how many *distinct* processes it has visited, by marking a process "visited" after visiting it.)

We use a more efficient strategy – the initial values of the variables *SNTM*, *RECM*, $SNTP_i$, $RECP_i$ are assigned in a different way than mentioned earlier. This assignment guarantees the following two conditions:

1. As long as there is at least one process that has not been visited yet, the condition *SNTM* = *RECM* will remain false, i.e., at least one pair of corresponding elements in the two arrays will not match. (We will be assuming that each process has at least one adjacent primary line. Otherwise we have isolated processes in the system. If needed, such cases can be incorporated in the scheme in obvious ways.) This is stated as lemma 6 below.

2. Moreover, this assignment does not affect the correctness of theorems 1 and 4. (In fact, all of lemmas 1, 3-5 and theorem 1-4 remain valid.) This is stated as lemma 7 below.

Obviously, this strategy is more efficient since the additional counter in the marker is avoided, reducing its length. An infinite set of assignments guaranteeing the above conditions exist; here we consider one specific assignment.

Corresponding to every primary line e = (i, j), we initialize *SNTM*(e) = 1, *RECM*(e) = 0, $SNTP_i(e) = 1$, and $RECP_j(e) = 2$. The marker declares termination after a visit if it finds that *SNTM* = *RECM*. The rest of the algorithm remains the same as before. Theorems 5 and 6 below prove the correctness of the algorithm.

Lemma 6: With the above initialization, suppose after visiting a process the marker finds that *SNTM* = *RECM*. Then, the marker has visited every process at least once by this time (say T).

Proof: For any line (i, j), we show that the marker has visited both i and j by the time T. (Since every process has at least one adjacent line, this establishes the result.) Suppose, to the contrary, this is not true for a line e = (i, j). Consider the following cases.

Case 1: The marker has not visited the process j by the time T. Obviously, in this case *SNTM*(e, T) $\geq$ 1 and *RECM*(e, T) = 0. This contradicts the assumption that *SNTM* = *RECM* at time T.

Case 2: The marker has visited process j, but not i, by the time T. Obviously, in this case *SNTM*(e, T) = 1, and *RECM*(e, T) $\geq$ 2. Again, this leads to a contradiction. This completes the proof.

Lemma 7: With the new initial values lemmas 1 and 3-5, and theorems 1-4 remain valid.

Proof: Note that with the new initial values, lemmas 1 and 2 remain valid. The results (5) and (6) in lemma 3 become slightly incorrect – the corrected versions of these results are:

$$tsnt(e,\ t) = SNTM(e,\ t) + SNTP_i(e,\ t) - 2 \tag{5'}$$

$$trec(e,\ t) = RECM(e,\ t) + RECP_j(e,\ t) - 2 \tag{6'}$$

The proofs of (5') and (6') are similar to the proofs of (5) and (6) before. From (5') and (6') it follows that lemmas 4 and 5 remain valid.

The previous proofs of theorems 1-4 do not directly rest on lemma 2 or the initial values of the program variables (so long as lemmas 1 and 3-5 remain valid). Therefore their correctness is not affected. This completes the proof.

Theorem 5: If the underlying computation is terminated at a time T_f, then the marker would declare termination within a finite time after T_f.

Proof: Follows from lemma 7 and theorem 1.

$\blacksquare$

Theorem 6: Suppose at a moment T, the marker declares termination. Then at this moment, the underlying computation is, indeed, in the terminated state.

Proof: The theorem follows from lemmas 6 and 7 and theorem 4.

$\blacksquare$

Some Improvements and Details

We briefly mention below some simple performance improvements to the algorithm. We also discuss a few details related to implementation.

1. Instead of keeping the two arrays *SNTM* and *RECM* in the marker, it is sufficient to keep a single array, say *SRM*, which would equal *SNTM* - *RECM*. This would reduce the secondary message length. Also, this reduces the chances of an overflow. (Elements of arrays *SNTM* and *RECM* are non-decreasing with time.)

2. In our description of the algorithm, the arrays $SNTP_i$ and $RECP_i$ have an element for every primary line of the network. Usually a process is connected to only a few other processes; in such cases, with this data structure updating *SNTM* or *RECM* or *SRM* may be quite inefficient. It may be more efficient to assign contiguous local line ids to the adjacent lines at each process, keep elements only for the adjacent lines in arrays $SNTP_i$ and $RECP_i$, and keep an array that maps from local line ids to global line ids.

3. How does the marker determine the next line to be traversed? If C is a simple cycle, then obviously just keeping the successor's id at each process is sufficient. Otherwise, one may keep a circular list of outgoing lines at each process (a line may be repeated several times in this list) and a local pointer that points to the next line to be followed by the marker. These circular lists can be initialized by considering a single traversal of the cycle C "by hand". The pointers can be initialized by defining the starting point of the

marker on the cycle. Note that the marker itself does not carry any information about its path of traversal; otherwise the secondary messages would become even longer.

Performance of the Algorithm

In the worst case, the number of message communications after the occurrence of termination is |C|-1. C can be chosen to be an elementary cycle, in which case this equals N-1, where N is the total number of processes in the system. Each message has a length of E integers, where E is the number of primary lines in the system. If communication delays depend significantly on the length of the messages, then this would be quite inefficient. On the other hand, if the message length does not significantly affect communication delays then this scheme would give a reasonable performance. One nice feature of this scheme is that in the *best* case, the number of message communications after termination is zero. Normally marker based algorithms [Misra 83, Chandy85a] require at least one complete cycle between the occurrence of termination and its detection.

5. Class 2 of Algorithms: Counting Total Number of Primary Messages Sent and Received in the System

Our motivation for devising algorithms in this class is to reduce the length of secondary messages. Here the marker has two *scalar* variables *SNTM2* and *RECM2* where it keeps its knowledge regarding the total number of primary messages sent and received, respectively, in the system. This differs from algorithms in class 1 where information about *individual lines* was being kept. Each process i has two scalar variables $SNTP2_i$ and $RECP2_i$. At any time $SNTP2_i$ = the total number of primary messages sent out by process i after the last visit of the marker at i (or since the initial time, if the marker has not visited i yet). $RECP2_i$ is similarly defined for messages received. The algorithm-skeleton of class 1 remains the same for this class, except that the variables *SNTM*, *RECM*, $SNTP_i$, and $RECP_i$ are replaced by *SNTM2*, *RECM2*, $SNTP2_i$, and $RECP2_i$ respectively. These variables are initialized to be zero, before the primary computation starts. (Unlike the discussion in class 1, we won't find a need to change this initialization later.) As in class 1, a process does not receive any messages during the interval between the start of the marker's visit and its departure. As before, the variables $SNTP2_i$ and $RECP2_i$ are incremented on sending or receiving (respectively) a primary message.

Now we consider the issue of when the marker declares termination. Theorem 7 below states that if the primary computation terminates at time T_f then within a finite time after T_f the system would reach a state where the condition $SNTM2 = RECM2$ will be true and will remain true forever afterwards. As before, we have to avoid the possibility of detecting "false termination". Again we ask the question – suppose in a sequence of visits along the cycle C, the marker continuously finds that $SNTM2 = RECM2$. Can it conclude termination after a (predefined) finite number V of such visits? Unfortunately, the answer in this case is in the negative, as shown in example 1 below. Theorem 8 below gives a method to complete the algorithm. Theorem 9 considers simple variations of the method given by theorem 8. These variations reduce the computational requirements at the processes; they do not improve communication requirements. Theorem 10 improves the performance of the algorithm by reducing the number of message communications after termination. After proving theorem 10, we show that certain simple and obvious variations of theorem 10 do not work. First let us discuss some intermediate results that will be used in the proofs.

Note that the variables $SNTM$, $RECM$, $SNTP_i$, and $RECP_i$ of class 1 can be used as "auxiliary" or "ghost" variables in our proofs. The notion of auxiliary variables is discussed, for example, in [Owicki 76]. The use of these auxiliary variables in our proofs is not essential; we use them only to simplify our proofs by exploiting the results in section 4. We assume that these variables are initialized to be zero at the start of primary computation.

Lemma 8: Lemmas 1-5 and theorems 1-4 of section 4 remain valid for the present algorithm-skeleton (when the variables $SNTM$, $RECM$, $SNTP_i$, and $RECP_i$ are interpreted as auxiliary variables).

Proof: In the algorithm-skeleton of section 4, let us introduce variables *SNTM2*, *RECM2*, $SNTP2_i$, and $RECP2_i$ in the same way as they are used in the present algorithm-skeleton. Obviously, the previous results of section 4 hold for this new algorithm-skeleton. Now in this algorithm-skeleton, let us treat variables *SNTM*, *RECM*, $SNTP_i$, and $RECP_i$ as auxiliary variables. Obviously, the results would still hold.

Lemma 9: At any time t,

$$SNTM2(t) = \text{sum } \{SNTM(e,\ t), \text{ over all primary lines e}\} \quad (9)$$

$$RECM2(t) = \text{sum } \{RECM(e,\ t), \text{ over all primary lines e}\} \quad (10)$$

$$SNTP2_i(t) = \text{sum } \{SNTP_i(e,\ t), \text{ over all outgoing primary lines e of process i}\} \quad (11)$$

$$RECP2_i(t) = \text{sum } \{RECP_i(e,\ t), \text{ over all incoming primary lines e of process i}\} \quad (12)$$

Proof: Obvious, by induction on the number of events in the system.

Lemma 10: At any time t,

$$\text{tr}(t) = SNTM2(t) - RECM2(t) + \text{sum } \{SNTP2_i(t), \text{ over all processes i}\} - \text{sum } \{RECP2_i(t), \text{ over all processes i}\} \quad (13)$$

where tr(t) = the total number of primary messages in transit at time t.

Proof: Let us take the sum of each side of (7) over e, e ranging over all primary lines in the system. The result follows from lemmas 9 and 8.

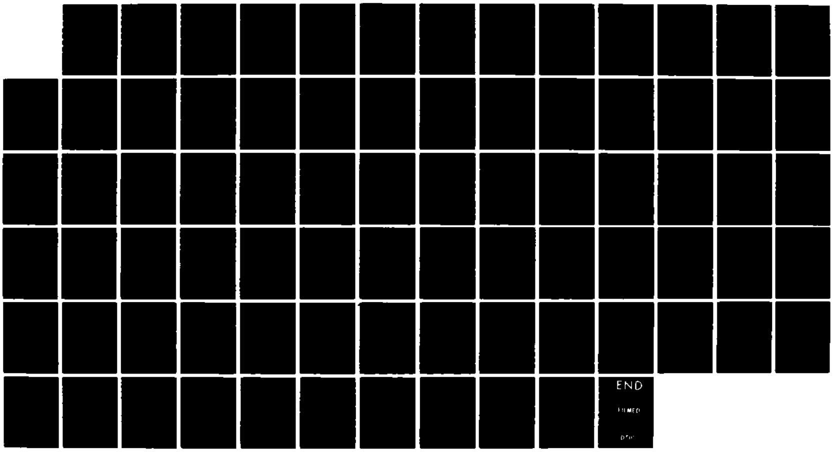
Theorem 7: Theorem 1 of section 4 remains valid if *SNTM* and *RECM* in that theorem are replaced by *SNTM2* and *RECM2* respectively.

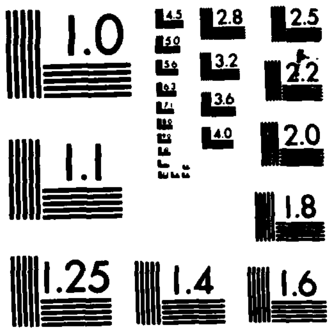
Proof: Follows from lemma 8 and the results (9) and (10) in lemma 9.

$\square$

AD-A159 189 RESEARCH TO STUDY SPECIFIC IMPORTANT PROBLEMS IN DISTRIBUTED SYSTEMS AND (U) TEXAS UNIV AT AUSTIN DEPT OF COMPUTER SCIENCES K M CHANDY ET AL. JUL 85 2/2

UNCLASSIFIED AFOSR-TR-85-0732 AFOSR-81-0205 F/G 9/2 NL





MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS-1963-A

Example 1: To show that there exist computations (following the algorithm-skeleton) where in an infinite sequence of visits, the marker continuously finds that *SNTM2* = *RECM2*, and yet the primary computation never terminates.

Consider a network of 10 processes. The cycle C is the elementary cycle 1, 2, ..., 10, 1. Initially the marker is at process 1, process 5 is active, and process 10 is idle. Processes 1-4 and 6-9 never send or receive a primary message and are always idle. Consider the following sequence of events at processes 5 and 10:

1. 5 sends a primary message to 10, 10 receives it, 10 sends a primary message to 5, 5 receives it. At this point 5 becomes idle and 10 remains active.

2. The marker visits 5, and departs.

3. 10 sends a primary message to 5, 5 receives it, 5 sends a primary message to 10, 10 receives it. At this point 10 becomes idle and 5 remains active.

4. The marker visits 10, and departs.

5. The above steps 1-4 are repeated indefinitely.

Obviously, after every visit the marker will find that *SNTM2* = *RECM2*. But the primary computation would never terminate!

The above example illustrates why after a finite number of visits with *SNTM2* = *RECM2* after each visit, the marker can not in general announce termination. Roughly speaking, a process i may have sent and received messages in between two successive visits by the marker. Theorem 8 is based on this observation.

Theorem 8: Suppose in a sequence of V = |C| visits, the marker continuously finds that $SNTP2_i = RECP2_i = 0$ before each visit (except possibly the first visit in the sequence) and *SNTM2* = *RECM2* after each visit. Then, at the end of this sequence of visits it can conclude that the underlying computation has terminated.

Proof: We show by induction on the number of visits in the sequence that after each visit *SNTM* = *RECM*. The result follows by theorem 3.

Base Case: Consider the first visit. Let T_0 be the time when the first visit in the sequence is completed. Obviously, $SNTP2_i(T_0) = RECP2_i(T_0) = 0$ for each process in the system. Therefore from (11) and (12) in lemma 9, $SNTP_i(e, T_0) = RECP_j(e, T_0) = 0$ for each primary line e = (i, j). Hence from lemmas 4 and 1, $SNTM(e, T_0) \geq RECM(e, T_0)$ for any e. But $SNTM2(T_0) = RECM2(T_0)$. Therefore from (9) and (10) in lemma 9, we get $SNTM(e, T_0) = RECM(e, T_0)$ for every primary line e. Therefore $SNTM = RECM$ at time T_0.

Inductive Case: Inductively, suppose $SNTM = RECM$ after the kth visit. By the hypothesis of the theorem, at the start of the (k+1)st visit $SNTP2_i = RECP2_i = 0$ where i is the process being visited. Using (11) and (12) in lemma 9 it follows that $SNTM = RECM$ at the end of the visit.

|||

Note: The above proof shows that if in a computation the hypothesis of theorem 8 is true then so is the hypothesis of theorem 3. The converse also follows, in an obvious way. Hence the two algorithms will require the same number of secondary message communications after and before the occurrence of termination. (Since the computation time in a visit in the two algorithms is different, the sequence of events in the two algorithms may be differer.. The above remark ignores any such differences.)

We state below some simple variations of theorem 8. These variations reduce only the processing requirements during a visit by the marker. Theorems 8 and 9 require the same number of secondary message communications after and before termination (again, this assumes that different processing requirements during a visit won't affect the sequence of events).

Theorem 9: Theorem 8 remains valid under any one of the following modifications (note: we are *not* considering here a combination of these modifications):

1. The requirement $SNTP2_i = RECP2_i = 0$ is replaced by $SNTP2_i = 0$.

2. The requirement $SNTP2_i = RECP2_i = 0$ is replaced by $RECP2_i = 0$.

3. The requirement "$SNTM2 = RECM2$ after each visit" is replaced by "$SNTM2 = RECM2$ at the end of the last visit of the sequence".

Proof: It is easy to see that any variation stated in theorem 9 is equivalent to theorem 8, in the sense that if the hypothesis of one is true then so is the hypothesis of the other.

$\blacksquare$

Now we consider a stronger modification to theorem 8. The algorithm suggested by theorem 10 below is *more* efficient than the one suggested by theorem 8, in terms of the number of message communications required after termination. We will discuss this after proving the theorem.

Theorem 10: Suppose in a sequence of $V = |C|$ visits, the marker continuously finds that $RECP2_i = 0$ before each visit (except possibly the first visit in the sequence) and $SNTM2 = RECM2$ at the completion of the last visit of the sequence. Then at the end of this sequence of visits it can be concluded that the underlying computation is terminated.

Proof: Let T_0 and T, respectively, be the times when the first and the last visits of the sequence were completed. Let t_i be the time when the marker completed its last visit at process i up to (including) time T. From (10),

$RECM2(T)$ = sum {$RECM$(e, T), over all primary lines e}

= sum {$trec$(e, t_j), over all primary lines e = (i, j)} by (6).

= sum {$trec$(e, T_0), over all primary lines e = (i, j)} since, obviously, in the interval $[T_0, t_i]$ process i did not receive any primary messages.

Similarly,

$$\mathit{SNTM2}(\mathrm{T}) = \text{sum } \{\mathit{tsnt}(\mathrm{e},\ \mathrm{t_i}), \text{ over all primary lines } \mathrm{e} = (\mathrm{i},\ \mathrm{j})\} \text{ by (9) and (5)}$$

$$= \text{sum } \{\mathit{tsnt}(\mathrm{e},\ \mathrm{T_0}), \text{ over e}\} + \text{sum } \{r'(\mathrm{e}), \text{ over e}\}$$

where $r'(\mathrm{e})$ = the number of primary messages sent on the line e = (i, j) during the interval $[\mathrm{T_0},\ \mathrm{t_i}]$.

Since $\mathit{SNTM2}(\mathrm{T}) = \mathit{RECM2}(\mathrm{T})$, using (2) we get $\mathit{tsnt}(\mathrm{e},\ \mathrm{T_0}) = \mathit{trec}(\mathrm{e},\ \mathrm{T_0})$ and $r'(\mathrm{e}) = 0$ for every primary line e. This is the same as conditions (B) and (C) in the proof of theorem 3. The rest of the proof is the same as the proof of theorem 3 after observation (C). (Alternatively, for any primary line e = (i, j), $\mathit{tsnt}(\mathrm{e},\ \mathrm{t_i}) - \mathit{trec}(\mathrm{e},\ \mathrm{t_j}) = \mathit{tsnt}(\mathrm{e},\ \mathrm{T_0}) + r'(\mathrm{e}) - \mathit{trec}(\mathrm{e},\ \mathrm{T_0}) = 0$. Hence by (8), $\mathit{SNTM}(\mathrm{e},\ \mathrm{T}) = \mathit{RECM}(\mathrm{e},\ \mathrm{T})$. The result follows from theorem 4.)

$\square$

Now we show that theorem 10 suggests a more efficient algorithm than theorem 8. Obviously, if the hypothesis of theorem 8 holds at a point in computation, then the hypothesis of theorem 10 holds as well. Example 2 below shows that the converse is not true. (However, for a given network topology, the worst case number of message communications after occurrence of termination is the same in both cases.)

Example 2: Consider the network of example 1 with the same initial conditions, except that the marker is initially at process 9. As before , processes 1-4 and 6-9 always remain idle. Consider the following sequence of events at processes 5 and 10:

1. 5 sends a message to 10, 10 receives it. At this point both processes are idle.

2. The marker arrives at process 10.

In the algorithm given by theorem 10, the marker will visit processes 10, 1, ..., 9 and then declare termination. Using the algorithm given by theorem 8, the marker will visit processes 10, ..., 5, ..., 10, ..., 4 and then declare termination.

One may be tempted to consider the following variation of theorem 10 – replace the requirement $RECP2_i = 0$ by $SNTP2_i = 0$. Example 3 below shows that this won't work.

Example 3: Consider the network of example 1 with the same initial conditions and the same behavior of processes 1-4 and 6-9. Consider the following sequence of events on processes 5 and 10:

1. 5 sends a primary message to 10 and becomes idle. (10 has not received it yet.)

2. Marker visits 5. It "restarts" a new sequence since $SNTP_5 \neq 0$ at the start of the visit.

3. Marker visits 10 and departs.

4. 10 receives the primary message sent by 5. It sends a primary message to 5 and remains active. 5 receives this message and remains idle.

5. Marker visits 5 and declares termination.

But process 10 is still active!

Since the above variation of theorem 10 doesn't work, it follows that the following variation will also not work – replace the requirement $RECP2_i = 0$ by ($RECP2_i = 0$ or $SNTP2_i = 0$). (If this variation had worked, obviously it would have been more efficient than theorem 10.)

We complete the algorithm-skeleton for class 2 by using theorem 10. Along the lines of the proof of theorem 1, it can be shown that if the primary computation terminates, say at time T_f, then the hypothesis of theorem 10 will become true within a finite time after T_f. The correctness of the algorithm follows from this observation and theorem 10.

Some Improvements and Details

1. As in class 1 (see "some improvements and details" in section 4), instead of keeping the two variables *SNTM2* and *RECM2* in the marker, it is sufficient to keep only a single variable *SRM2* which would equal *SNTM2* - *RECM2*. This has the same advantages as before.

2. Also, at a process i instead of keeping $SNTP2_i$ and $RECP2_i$, one may keep a variable $SRP2_i$ which would equal $SNTP2_i$ - $RECP2_i$, and a boolean variable to indicate if $RECP2_i = 0$. This, of course, does not improve the efficiency regarding message communications; it only reduces the processing time involved in a visit.

3. Same as 3 in our discussion under "some improvements and details" for class 1.

4. How does the marker detect that the first condition of theorem 10 holds for the entire sequence? We roughly sketch a few possible ways of doing this:

   a. Sequence Length Counter: The marker carries a counter for this purpose. Initially this counter is 0. On visiting a process i, if $RECP2_i$ is zero at the start of the visit then the counter is incremented; else it is reset to 1. After the visit if the counter is $\geq |C|$, then the condition *SNTM2* = *RECM2* is checked.

   What if the counter has become $\geq |C|$ and the condition *SNTM2* = *RECM2* is not met? If the counter keeps getting incremented indefinitely, it may overflow. To avoid this, one may reset the counter to 1 during a visit if it is $\geq |C|$ at the start of the visit. This raises the following issue: there seems to be a possibility that termination may be detected after "too many" visits. For example, what if the condition $RECP2_i = 0$ is true before every visit, but the condition *SNTM2* = *RECM2* becomes true after $|C|+1$ visits and the counter was reset to 1 (to avoid overflow) during the visit $|C|+1$? It can be shown that such cases can not arise. In other words, at the start of a visit if the counter is $\geq |C|$, then it can be reset to 1 without loss of efficiency. At any such point T it can be asserted that the marker will definitely visit (either in future or in current visit) a process i such that $RECP2_i \neq 0$ at the start of the visit. To prove this, suppose this is not true. Then there are two possibilities: (i) There is a finite sequence of visits made after time T such that $RECP2_i = 0$ before each such visit at the process i being visited and *SNTM2* = *RECM2* at the completion of the last visit of the sequence. (ii) There is an infinite sequence of visits made after time T such that

$RECP2_i = 0$ before each visit and $SNTM2 \neq RECM2$ after each visit. Note that after time T, for any visit at a process i if $RECP2_i = 0$ before the visit then $SNTP2_i = 0$ before the visit as well. Since $SNTM2(T) \neq RECM2(T)$, (i) above is impossible. In case (ii), obviously we have primary messages in transit at time T. Within a finite time one of these messages will be received at a process i, making $RECP2_i$ nonzero. Hence (ii) above is impossible. This completes the proof.

b. Round Number: For simplicity, first let us assume that C is an elementary cycle. The marker contains a round number. At the start of a visit (say at process i) if $RECP2_i \neq 0$ then a new round is started, i.e., marker's round number is incremented. During any visit at a process i, the marker's round number is stored in a local variable at i. If at the start of a visit, the round number of the marker equals that of the current process i, it means that the marker has previously made a sequence of at least |C| visits such that before each visit (except possibly the first one) $RECP2_j = 0$ at the corresponding process j. Therefore in this case if $RECP2_i = 0$ at the start of the visit, the condition $SNTM2 = RECM2$ is checked for termination after completion of the visit. (Alternatively, the termination check could be made at the start of a visit if the round numbers match.) At the start of secondary computation, round numbers of the marker and the processes are initialized to 1 and 0 respectively.

If the round number of the marker keeps getting incremented indefinitely, it may overflow. To solve this problem one may increment the round number as 1+ [(round number) mod |C|]. The new round number generated would obviously be different from local round numbers of all processes (except possibly the one being visited).

(As a side note, using this method the number of message communications after occurrence of termination is increased by 1.)

If C is not an elementary cycle, a counter may be kept at each process that counts the number of times the process has been visited in the current round.

c. Initial Process Id: Again let us first assume that C is an elementary cycle. In this method the marker keeps a pointer that points to the process id of the first process in the current sequence of visits such that before each visit (except possibly the first one) $RECP2_i = 0$. At the start of a visit (say at process i) if $RECP2_i \neq 0$ then this pointer is set to i. If at the start of a visit, this pointer is pointing to the current process i, this means that the marker has previously made a sequence of at least |C| visits such that before each visit (except possibly the first one) $RECP2_j = 0$ at the corresponding process j. Therefore in this case if $RECP2_i = 0$ at the start of the visit, then the condition *SNTM2* = *RECM2* is checked for termination after completion of the visit. Similar to our discussion in (a) above (using sequence length counter), if the condition *SNTM2* = *RECM2* is false in this check, we need not reset the pointer.

Obviously, there is no overflow problem in this approach. As in the method using round numbers, the number of messages after termination in this method is increased by 1. If C is not an elementary cycle, one may keep local counters at the processes to count the number of times the process currently pointed to by the marker has been visited in the current sequence.

5. Suppose we designate a specific process where the decision regarding termination would be taken. In this case the marker needs to carry only an integer (the value of *SNTM2* - *RECM2*) and a boolean (instead of an integer as in 4 above) which remembers whether in the current round the first condition of theorem 10 has been true so far. Since message length has decreased, this improves the performance in the worst case. However, in the average case the number of message communications after termination will increase.

Performance of the Algorithm

The worst case occurs when the marker departs a process i and before it reaches the next process, process i receives a primary message and the primary computation terminates at this point. The number of secondary messages sent after the termination of primary computation in this case is 2.|C| - 2. Each secondary message contains two integers – one integer containing the value *SNTM2* - *RECM2*, and the other used to check the first condition of theorem 10, as discussed in 4 above under "some improvements and details".

6. Class 3 of Algorithms: Using Multiple Markers

In classes 1 and 2 we have a *single* marker that sequentially traverses the system. In this section we will use multiple markers to enhance performance. First we observe that the *sequential* traversal of the system by a marker in the previous algorithms is not essential. If several processes could be visited in parallel, even then these results will hold. The following theorem is obtained from theorem 10 by an abstraction of the proof of that theorem (i.e., by avoiding details regarding sequential nature of the traversals). The proof of this theorem is essentially the same as that for theorem 10.

Theorem 11: Let $[T_0, T]$ be a time interval during which several visits have been completed, possibly in parallel. Suppose these visits satisfy the following:

1. At least one visit is completed at each process during this interval (the start times of these visits need not be in the interval).
2. At the start of each visit, say at process i, $RECP2_i = 0$, and
3. At time T $SNTM2 = RECM2$.

Then, at time T the primary computation is terminated.

Notes:

1. The values of *SNTM2* and *RECM2* at time T are defined in the obvious way – the results of various visits have to be accumulated.
2. Theorem 10 follows as a special case of theorem 11. On first sight this might not be so obvious, since theorem 10 allows the value of $RECP2_i$ for the first visit to be nonzero. However, after the very first visit in the sequence, one may consider an imaginary visit to the same process – theorem 10 would then readily follow from theorem 11.

Using theorem 11, one may devise schemes using several markers. The markers would check the values of $RECP2_i$, and accumulate values for *SNTM2* and *RECM2* in different parts of the system (these parts need not be disjoint).

A. Using Two Markers

Let us assume that we have two paths P_1 and P_2 from a given process I to a given process J. Also assume that these paths together cover all the processes in the system. Initially both markers are kept at process I. Then they traverse the two paths respectively. After both have visited J, a check for termination is made as follows. The values *SNTM2* and *RECM2* are computed by adding the corresponding values in the two markers. Each marker i also has a boolean variable $NZREC_i$ which is set to true if at the start of some visit at a process j in the current traversal of the path, $RECP2_j$ was found to be nonzero. At J, if *SNTM2* = *RECM2* and both booleans are false then termination is announced. Otherwise a new traversal is to be started. To start a new traversal, both markers may be sent back to I via a line (J, I). Alternatively, the markers may traverse the paths P_1 and P_2 in the reverse direction in which case the next check for termination would be made at process I.

Now we make a simple modification to the above scheme that would lead to an obvious generalization for the case of more than two markers. A new process, called a *central process* (*CP*), is introduced in the system where the check for termination would be made. (This process may be *implemented* as part of some existing process in the system.) Paths P_1 and P_2 now need not share their initial and final processes. Initially both markers are at the CP. A traversal of the system is started by the CP, by sending the markers to the initial processes of the respective paths. After traversing the paths, the markers arrive at the CP where the decision regarding termination is made in the same way as before.

Now we consider an erroneous variation of this scheme which supposedly attempts to improve its efficiency. Suppose a marker i has arrived at the CP after traversing its path and $NZREC_i$ is true. Suppose the other marker has not yet arrived at the CP. One might be tempted to consider the following. Since marker i knows that termination can not be announced after this traversal, it doesn't wait for the other marker to arrive; instead it goes back to traverse P_i. Equivalently, a marker i would traverse its path P_i repeatedly until the value of $NZREC_i$ is false at the end of a traversal, and then it would go to CP and wait for a termination check to be made. The following simple example shows that this scheme won't work:

Example 4: Let P_1 and P_2 consist of single processes, processes 1 and 2 respectively. Initially process 2 is idle and process 1 is active. Consider the following sequence of events:

1. Marker 2 visits process 2. It departs from process 2 (but hasn't arrived at the CP yet).

2. Process 1 sends a primary message to process 2. Process 2 receives this message and sends another one to process 1. Process 1 receives it and becomes idle. Process 2 remains active.

3. Marker 1 visits process 1. Since the value of $NZREC_1$ is true after this visit, it visits process 1 again (equivalently, after the first visit it goes to CP, then goes back and visits process 1). Now it arrives at the CP.

4. Marker 2 arrives at CP. Obviously both booleans $NZREC_i$ are false and $SNTM2 = RECM2$ at this point. Hence termination is declared. But process 2 is still active!

Performance of the Scheme

Let us assume that the length of each path P_1 or P_2 is approximately N/2. For worst case, consider the following scenario. Marker 1 visits and departs from the first process (say i) on its path. Now process i receives a primary message and at this point the primary computation is terminated. Obviously termination won't be detected after the current traversal. Again, it won't be detected in the next traversal since $RECP2_i$ would be nonzero at the start of the next visit to i (let us assume that i appears only once on P_1, and doesn't appear on P_2; otherwise this won't be strictly true). So the number of secondary message communications after termination is $\approx 3N/2$. Each such message consists of an integer and a boolean.

B. Using More Markers

One may similarly use a CP, K paths, and K markers in general. Let L be the length of the longest of these paths. By considering a scenario similar to the above, we have the worst case number of message communications $= 3L + 4$. If each path has N/K processes then this equals 3N/K + 1. Note that as K is increased, the scheme tends to become more centralized. With K = N it is a purely centralized scheme (i.e., each process interacts only with the central processor for termination detection) with worst case number of message communications after termination = 4.

7. Conclusion

We have presented a class of efficient algorithms for termination detection in distributed systems. Our assumptions regarding the underlying computation are simple. In particular we do not require the FIFO property for the communication channels. Also, the topological requirements about communication paths are simple and flexible, both from the correctness and performance point of view. We discussed the correctness and performance of our algorithms. Depending upon the application, the nature of the chosen algorithm can be varied incrementally from a distributed one to a centralized one.

We introduced message counting as an effective technique in designing termination detection algorithms. We showed how one can avoid counting messages for *each and every line*, normally resulting in better performance. Our presentation involves deriving algorithms via a sequence of simple modifications. Several correct as well as incorrect variations have been considered. We hope that this approach of presentation has resulted in better understandability of the algorithms.

Acknowledgments

I am indebted to Professors J. Misra and K. M. Chandy for their masterful teaching of several courses at UT Austin which have significantly influenced me. Their financial support for this work is greatly appreciated. The idea of counting messages on the lines was inspired by a similar thought of Professor K. M. Chandy. I am thankful to Professor J. Misra for his encouragement and for his thoughts on performance of termination detection algorithms. Professor M. Gouda's encouragement and his positive influences on my writing style are greatly appreciated. I would like to extend my special thanks to Ted Briggs and Pradeep Jain for their comments on earlier drafts of this paper.

References

[Beeri 81] C. Beeri and R. Obermarck, "A Resource Class Independent Deadlock Detection Algorithm", *Research Report RJ3077*, IBM Research Laboratory, San Jose, California, May 1981.

[Bracha 83] G. Bracha and S. Toueg, "A Distributed Algorithm For Generalized Deadlock Detection", *Technical Report TR 83-558*, Cornell University, June 1983.

[Chandy 81] K. M. Chandy and J. Misra, "Asynchronous Distributed Simulation Via a Sequence of Parallel Computations", *Communications of the ACM*, Vol. 24, No. 4, pp.198-205, April 1981.

[Chandy 82a] K. M. Chandy and J. Misra, "A Distributed Algorithm for Detecting Resource Deadlocks in Distributed Systems", *ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing*, Ottawa, Canada, August 1982.

[Chandy 82b] K. M. Chandy and J. Misra, "A Computation on Graphs: Shortest Path Algorithms", *Communications of the ACM*, Vol. 25, No. 11, pp.833-837, November 1982.

[Chandy 83] K. M. Chandy, J. Misra, and L. Haas, "Distributed Deadlock Detection", *ACM Transactions on Computing Systems*, Vol. 1, No. 2, pp. 144-156, May 1983.

[Chandy 85a] K. M. Chandy and J. Misra, "A Paradigm for Detecting Quiescent Properties in Distributed Computations", working paper, Department of Computer Sciences, University of Texas, Austin, Texas 78712, January 9, 1985.

[Chandy 85b] K. M. Chandy and L. Lamport, "Distributed Snapshots: Determining Global States of Distributed Systems", to appear in *ACM Transactions on Computing Systems*.

[Chang 82] E. Chang, "Echo Algorithms: Depth Parallel Operations on General Graphs", *IEEE Transactions on Software Engineering*, Vol. SE-8, No. 4, pp.391-401, July 1982.

[Cohen 82] S. Cohen and D. Lehmann, "Dynamic Systems and Their Distributed Termination", *ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing*, pp. 29-33, Ottawa, Canada, August 18-20, 1982.

[Dijkstra 80] E. W. Dijkstra and C. S. Scholten, "Termination Detection for Diffusing Computations", *Information Processing Letters*, Vol. 11, No. 1, August 1980.

[Dijkstra] E. W. Dijkstra, "Distributed Termination Detection Revisited", EWD 828, Plataanstraat 5, 5671 AL Nuenen, The Netherlands.

[Francez 80] N. Francez, "Distributed Termination", *ACM Transactions on Programming Languages and Systems*, Vol. 2, No. 1, pp. 42-55, January 1980.

[Francez 81] N. Francez, M. Rodeh, and M. Sintzoff, "Distributed Termination with Interval Assertions", *Proceedings of Formalization of Programming Concepts*, Peninusla, Spain, April 1981. Lecture Notes in Computer Science 107, (Springer-Verlag).

[Francez 82] N. Francez and M. Rodeh, "Achieving Distributed Termination Without Freezing", *IEEE-TSE*, Vol. SE-8, No. 3, pp.287-292, May 1982.

[Gligor 80] V. Gligor and S. Shattuck, "On Deadlock Detection in Distributed Data Bases", *IEEE-TSE*, Vol. SE-6, No. 5, September 1980.

[Haas 83] L. Haas and C. Mohan, "A Distributed Deadlock Detection Algorithm for a Resource Based System", *Research Report RJ3765*, IBM Research Laboratory, San Jose, California, January 1983.

[Herman 83] T. Herman and K. M. Chandy, "A Distributed Procedure to Detect AND/OR Deadlock", Department of Computer Sciences, University of Texas, Austin, 78712, February 1983.

[Hoare 78] C. A. R. Hoare, "Communicating Sequential Processes", *Communications of the ACM*, Vol. 21, No. 8, pp. 666-677, August 1978.

[Holt 72] T. Holt, "Some Deadlock Properties of Computer Systems", *Computing Surveys*, Vol. 4, No. 3, pp. 179-196, September 1972.

[Kumar 85] D. Kumar, "Distributed Simulation", Ph.D. Thesis (in preparation), Department of Computer Sciences, University of Texas, Austin, Texas 78712.

[Lamport 78] L. Lamport, "Time, Clocks, and the Ordering of Events in a

Distributed System", *Communications of the ACM*, Vol. 21, No. 7, July 1978.

[Misra 81] J. Misra and K. M. Chandy, "Proofs of Networks of Processes", *IEEE Transactions on Softaware Engineering*, Vol. SE-7, No. 4, pp. 417-426, July 1981.

[Misra 82a] J. Misra and K. M. Chandy, "Termination Detection of Diffusing Computations in Communicating Sequential Processes", *ACM Transactions on Programming Languages and Systems*, Vol. 4, No. 1, pp. 37-43, January 1982.

[Misra 82b] J. Misra and K. M. Chandy, "A Distributed Graph Algorithm: Knot Detection", *ACM Transactions on Programming Languages and Systems*, Vol. 4, No. 4, pp. 678-688, October 1982.

[Misra 83] J. Misra, "Detecting Termination of Distributed Computations Using Markers", *Proceedings of the ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing*, Montreal Canada, August 17-19, 1983.

[Obermarck 80] R. Obermarck, "Deadlock Detection For All Resource Classes", *Research Report RJ2955*, IBM Research Laboratory, San Jose, California, October 1980.

[Obermarck 82] R. Obermarck, "Distributed Deadlock Detection Algorithm", *ACM Transactions on Database Systems*, Vol. 7, No. 2, pp.187-208, June 1982.

[Owicki 76] S. Owicki and D. Gries, "An Axiomatic Proof Technique for Parallel Programs I", *Acta Informatica*, Vol. 6, pp.319-340, 1976.

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER	2. GOVT ACCESSION NO.	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) Manuscript: "A Model and Proof System for Asynchronous Networks"		5. TYPE OF REPORT & PERIOD COVERED final: 6/14/81 - 6/15/85
		6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s) Dr. Bengt Jonsson currently at Uppsala University		8. CONTRACT OR GRANT NUMBER(s) AFOSR 81-0205
9. PERFORMING ORGANIZATION NAME AND ADDRESS Computer Sciences Department University of Texas at Austin Austin, Texas 78712		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
11. CONTROLLING OFFICE NAME AND ADDRESS Capt. A. L. Bellamy AFOSR/NM Bolling AFB, DC 20332		12. REPORT DATE July 1985
		13. NUMBER OF PAGES
14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office)		15. SECURITY CLASS. (of this report)
		15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

to appear in the Proceedings of the 4th ACM Conference on the Principles of Distributed Computing, Minaki, Canada, August 5-7, 1985

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

We present a compositional model for nondeterministic asynchronous networks, which represents both safety and liveness properties. A network is represented by the set of its quiescent traces. A quiescent trace is the sequence of communication events in a computation, after which the network will not produce more output unless it receives more input. The representation of a network is derived from an operational definition of its behavior, in the form of a labeled transition system. Rules for composition, abstraction and renaming in the model are proven from their operational definitions, showing that the model is compositional. A method to specify networks in predicate logic is presented, together with a proof system. The method is demonstrated on the specification and verification of the alternating bit protocol.

A MODEL AND PROOF SYSTEM FOR ASYNCHRONOUS NETWORKS

Bengt Jonsson
Uppsala University, Institute of Technology*
S-751 21 Uppsala, Sweden

Abstract: We present a compositional model for nondeterministic asynchronous networks, which represents both safety and liveness properties. A network is represented by the set of its quiescent traces. A quiescent trace is the sequence of communication events in a computation, after which the network will not produce more output unless it receives more input. The representation of a network is derived from an operational definition of its behavior, in the form of a labeled transition system. Rules for composition, abstraction and renaming in the model are proven from their operational definitions, showing that the model is compositional. A method to specify networks in predicate logic is presented, together with a proof system. The method is demonstrated on the specification and verification of the alternating bit protocol.

1. Introduction

For deterministic asynchronous networks Kahn [K] has presented an elegant model which represents a process by a function from histories on input channels to histories on output channels. For nondeterministic networks, Brock and Ackerman [BA] have shown that channel histories are not adequate in a compositional model, since channel histories of the component processes do not provide enough information to calculate the channel histories of the whole network. Several solutions have been presented ([BA], [Br], [Pa], [Pr]), but it is not clear which one is easiest to use in practice.

*: Part of this work was carried out while the author was on leave at Stanford University.

This work was supported in part by the Swedish Board for Technical Development (STU) under Contract 82-3406, by the National Science Foundation under grant MCS-82-14523, by Defense Advanced Research Projects Agency under Contract N00039-84-C-0211, and by the United States Air Force Office of Scientific Research under Contracts AFOSR-81-0014 and AFOSR-81-0205.

to appear at
4th ACM Symposium on
Principles of Distributed Computing
Minaki, Canada, August, 1985

Compositionality can be attained in a simple way by using traces, i.e. totally ordered sequences of communication events on all channels of a process or a network (e.g. [BM], [CH], [MC]), but it is not evident how to represent liveness properties of nondeterministic networks.

Consider for example the buffer process in Fig. 1 that reads *a*'s on the left channel and for each *a* outputs a *b* on the right channel.

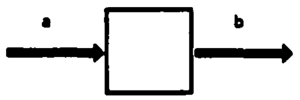


Figure 1. A buffer process

The traces of the process are the sequences of *a*'s and *b*'s with at least as many *a*'s as *b*'s. The set of traces, however, does not say anything about liveness properties, since another process, that at some nondeterministically chosen moment stops producing output, will have the same set of traces and yet have different liveness properties.

The idea of *quiescence*, due to Chandy and Misra [Mis], is a solution to the problem of representing liveness. The remainder of the introduction contains a summary of this idea and an outline of its further development in this paper.

A process or a network is in a quiescent state iff it remains inactive and does not produce any more output, unless it receives more input. A *quiescent trace* is the sequence of communication events in a computation after which the process becomes quiescent, i.e. a maximal trace that will not be extended unless more input is supplied. The finite quiescent traces of the buffer process in Fig. 1 are the traces that contain an equal number of *a*'s and *b*'s.

Since some processes may not terminate, all infinite traces are considered quiescent: an infinite trace can not be extended. As an example consider the source process in Fig. 2 that repeatedly sends *a* and never terminates.

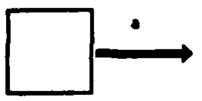


Figure 2. A source process

The process never becomes quiescent, but can be represented by the infinite trace $< aaa \ldots >$, denoted as a^{ω}.

Infinite traces make it possible to model fairness properties by requiring that each infinite trace represents a fair computation. For instance, the infinite traces of the buffer process in Fig. 1 contain an infinite number of b's.

A process or a network is represented by the set of its quiescent traces. A trace of a network is quiescent iff each projection of the trace onto a process of the network is quiescent, since a network is in a quiescent state iff each of its processes is in a quiescent state. Thus the representation is compositional. Safety properties correspond to the (possibly nonquiescent) traces of the network. These are obtained as prefixes of the quiescent traces. Liveness properties correspond to the quiescent traces: a nonquiescent trace will always be extended to a quiescent trace.

In this paper we develop the idea of quiescence in more detail. The main contributions are:

- A model for nondeterministic asynchronous networks, which represents a network by its quiescent traces. The model is derived from an operational definition in the form of labeled transition systems. From the operational definition we prove rules for composition, abstraction and renaming of networks in the model, showing that the model is compositional.
- A method for specifying and verifying both *safety* and liveness properties of asynchronous networks. Properties of quiescent traces are stated in predicate logic. We believe that a specification method based on the concise model will allow short specifications and proofs. The method is demonstrated on the specification and verification of the alternating bit protocol.

A related approach to the specification of liveness properties [MCS] uses a special condition corresponding to "nonquiescence". Another way to state liveness properties is to use temporal logic [Ha], [NGO]. Compositional models that use traces and represent liveness properties for synchronous networks [BHR], [FLP], [Mil], [NGO] must include information about which communication events the network is ready to perform.

The paper is organised as follows: In section 2 we present our view of networks, and in section 3 the operational semantics. A more elaborate discussion of the operational semantics is deferred to the appendix. In section 4, quiescent traces are defined and the model for representing networks is presented. Rules for composition, abstraction and renaming are given. In section 5 we present a method for specifying networks, followed by a proof system in section 6. The method is illustrated by a specification and verification of the alternating bit protocol in section 7.

2. Networks

A *network* in our model consists of a (finite) set of *processes* that are connected by uniquely named unidirectional *channels*. A channel that connects two processes of the network is of type *internal*. A channel that connects a process with the environment of the network is of type *external input* or *external output* channel, depending on its direction. The *signature* of a network is the set of channels together with their types. Fig. 3 depicts a network with three processes, whose signature contains three internal channels, two external input channels, and one external output channel.

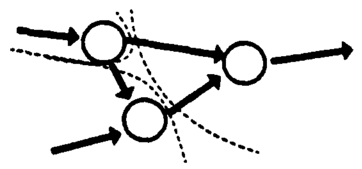


Figure 3. A network

Processes communicate by sending messages over channels. Our model is based on the following requirements.

1) A process can send output without cooperation by the receiver, since communication is asynchronous.
2) Communication events, such as the transmission of a message, should be atomic in order to simplify the model.

These requirements are satisfied by partitioning the network into subnetworks as shown by the dashed lines in Fig. 3. It is always possible to send a message into the input channel of another subnetwork. Communication between subnetworks corresponds to an atomic event, namely the passing of a message over a dashed line (i.e. a sending event). In the following we will therefore consider subnetworks, consisting of a process and its input channels, as basic entities.

The sending of the message m over the channel c is a *communication event* denoted by the pair (c, m). The event is called an internal event, an external input event, or an external output event, depending on the type of the channel c. The set of communication events of the network N is denoted E_N. We also consider *silent events* within a network. All silent events are denoted by τ. Examples are reading of messages from input channels, and communication on channels that are hidden as the result of an abstraction operation (defined below).

The following operations on networks will be studied:

- *Composition* of several networks $N_1, \ldots, N_k$, denoted by $N_1 \| \cdots \| N_k$. External output channels of the networks $N_1, \ldots, N_k$ are linked to external input channels with the same name and become internal channels of the network $N_1 \| \cdots \| N_k$.
- *Abstraction* of a set C of internal channels of a network N, denoted by $N \setminus C$. The channels in C disappear from the signature of $N \setminus C$. Communication events on channels in C become silent events.
- *Renaming* of a set of channels of the network N by a renaming function Φ, denoted by $N[\Phi]$. The function Φ is a bijection on the set of channel names that preserves their types.

3. Operational Semantics

As an operational definition of the behavior of a network N we use a labeled transition system. This is a quintuple $< \Sigma_N, E_N, R_N, \sigma_N^0, \mathcal{F}_N >$, where

Σ_N is a set of *configurations*.

E_N is the set of *communication events* possible on N's channels.

R_N is a set of *labeled transitions* in $\Sigma_N \times (E_N \cup \{\tau\}) \times \Sigma_N$.

σ_N^0 is the *initial* configuration.

$\mathcal{F}_N$ is a finite collection of *fairness sets*.

A configuration of N typically includes the states of the processes and the contents of the channels. Configurations of the network N are denoted by σ_N, σ'_N, etc.

A transition corresponds to the occurrence of an event and a simultaneous change of configuration. A transition from the configuration σ_N to the configuration σ'_N labelled by the event e, written as

$$\sigma_N \xrightarrow{e} \sigma'_N,$$

states that in the configuration σ_N the event e can occur and as a result the configuration changes to σ'_N. Here $e \in E_N \cup \{\tau\}$.

Note that it is not the case that every transition system models a network (cf. proposition 4.2. below). Further details about this, the modeling of fairness, and the definition of composition, abstraction and renaming in transition systems are found in the appendix.

4. The Model

4.1. Quiescent Traces

The notions of trace and quiescent trace are defined from the operational semantics in the following way:

- A *transition sequence* of a network N is a (finite or infinite) sequence of transitions in R_N, starting in the initial configuration. It is written

$$\sigma_N^0 \xrightarrow{e_1} \sigma_N^1 \xrightarrow{e_2} \cdots \xrightarrow{e_n} \sigma_N^n \xrightarrow{e_{n+1}} \cdots$$

with $e_i \in E_N \cup \{\tau\}$. (For infinite sequences there are certain fairness requirements, elaborated in the appendix.)

- A *trace* of the network N is the sequence of communication events (thus skipping τ's) in a finite or infinite transition sequence.
- A trace of N is *divergent* iff it is finite and consists of the communication events in an infinite transition sequence (which thus must end with an infinite sequence of τ-transitions).
- A trace of N is *quiescent* iff either
  1) It is the sequence of communication events in a finite transition sequence $\sigma_N^0 \xrightarrow{e_1} \cdots \xrightarrow{e_n} \sigma_N^n$ in which the only transitions possible from the last configuration σ_N^n are labeled by external input events.
  2) It is infinite.
  3) It is divergent.
- A quiescent trace of N is *nondivergent* iff either 1) or 2) above holds.

Note that a quiescent trace of a nondeterministic network can be both divergent and nondivergent, since there can be many transition sequences corresponding to the same trace.

4.2. The Model

In the model a network N is represented by the set of its quiescent and divergent traces, written as

$$[N] = \{q \mid q \text{ is a nondivergent quiescent trace of } N\} \cup \{q\uparrow \mid q \text{ is a divergent trace of } N\}$$

In the following we use q to denote the sequence of communication events in a quiescent trace. The symbol $\uparrow$ that is appended to divergent traces indicates that the computation does not terminate although only a finite number of events are observed.

The following proposition shows that the representation $[N]$ of a network N characterises both its safety properties (since all traces can be obtained as prefixes of the quiescent ones), and its liveness properties (a trace that is not quiescent will be extended to a quiescent trace).

Proposition 4.1. *A sequence of communication events is a trace of a network N iff it is the prefix of a (possibly divergent) quiescent trace of N.*

The proof is omitted. It follows from the definitions in section 4.1 and properties of transition systems.

□

Proposition 4.2. *The following properties always hold for the representation* $[N]$ *of a network* N

1) $[N] \neq \emptyset$
2) If the sequence of communication events t is a prefix of an element of $[N]$, then for each external input event $i \in E_N$ the sequence of events ti is a prefix of an element of $[N]$.
3) If the sequence of communication events t is a prefix of an element of $[N]$, then there is a quiescent trace in $[N]$ that extends t without using external input events.

The proof is omitted. It follows from the operational semantics in the appendix. Intuitively, 2) states that a network can always receive input, and 3) states that in each situation the network will continue to perform output and internal events until it reaches a quiescent state.
□

4.3. Operations on the Model

We show that the model is compositional by establishing rules corresponding to the composition, abstraction and renaming operations. The definition of these operations on the operational semantics (in the appendix) is used to prove the following rules.

Theorem 4.3. *Composition, abstraction and renaming correspond to the following operations in the model:*

Composition: Here q ranges over sequences of communication events in $\cup_i E_{N_i}$, and $\pi_i(q)$ denotes the projection of q onto the channels of N_i.

$$\begin{aligned}
&[N_1 \| \cdots \| N_k] = \\
&\{q \mid \text{ for all } i \ \pi_i(q) \in [N_i]\} \\
&\cup \left\{ q \;\middle|\; \begin{pmatrix} q \text{ infinite and} \\ \text{for all } i \ (\ \pi_i(q) \in [N_i] \text{ or } \pi_i(q) \uparrow \in [N_i]) \end{pmatrix} \right\} \\
&\cup \left\{ q\uparrow \;\middle|\; \begin{pmatrix} q \text{ finite and} \\ \text{for all } i \ (\ \pi_i(q) \in [N_i] \text{ or } \pi_i(q) \uparrow \in [N_i]) \\ \text{for some } j \ \ (\pi_j(q) \uparrow \in [N_j]) \end{pmatrix} \right\}
\end{aligned}$$

Abstraction: Here $q \backslash C$ denotes the result of deleting events on channels in C from q.

$$\begin{aligned}
&[N \setminus C] = \\
&\quad \{q \setminus C \mid \ q \in [N] \text{ , } q \text{ finite } \} \\
&\cup \{q \setminus C \mid \ q \in [N] \text{ , } q \text{ and } q \setminus C \text{ infinite } \} \\
&\cup \{(q \setminus C) \uparrow \mid \ q \in [N] \text{ , } q \text{ infinite and } q \setminus C \text{ finite } \} \\
&\cup \{(q \setminus C) \uparrow \mid \ q \uparrow \in [N]\}
\end{aligned}$$

Renaming: Here Φ denotes the pointwise extension of the renaming function Φ to traces, i.e. Φ renames the events of a sequence as follows: $\Phi((c,m)) = (\Phi(c), m)$.

$$\begin{aligned}
[N[\Phi]] = \ & \{\Phi(q) \ \mid \ q \in [N]\} \\
& \cup \{\Phi(q) \uparrow \mid \ q \uparrow \in [N]\}
\end{aligned}$$

Below we outline proofs of the rules.

Composition: The network $N_1 \| \cdots \| N_k$ is in a quiescent state iff each subnetwork is in a quiescent state. It follows that if q is a finite nondivergent quiescent trace of $N_1 \| \cdots \| N_k$ then the projection of q onto each subnetwork N_i is a finite nondivergent quiescent trace, since it corresponds to the part of the computation in which N_i is involved. If q is infinite, its projections onto the subnetworks must be (possibly divergent) quiescent traces. If q is divergent, it must have a divergent projection onto some subnetwork.

Conversely, if the projections of a sequence of communication events q onto all subnetworks are quiescent, then q is a quiescent trace of $N_1 \| \cdots \| N_k$. If one or more of the projections are divergent, then q will be divergent iff it is finite.

Abstraction: A network N is in a quiescent state iff $N \setminus C$ is in a quiescent state. For each transition sequence of N there is a transition sequence of $N \setminus C$ in which events on channels in C are changed to τ's. A quiescent trace q' of $N \setminus C$ is therefore obtained by deleting events on channels in C from a quiescent trace q of N. The trace q' is divergent either if q is divergent, or if an infinite τ-sequence is created in the corresponding transition sequence by deleting events, in which case q is infinite.

Conversely, a sequence of events obtained by deleting events from a quiescent trace q of N is a quiescent trace q' of $N \setminus C$. If q is divergent, then q' will also be divergent. If q is infinite but q' finite, then q' is divergent, since it corresponds to a transition sequence ending in an infinite τ-sequence.

Renaming: The rule simply states that the events of the quiescent traces of N should be renamed to obtain the quiescent traces of $N[\Phi]$. The proof is straight-forward.
□

Note that the composition operator is "fair" with respect to each subnetwork. As an illustration, consider the network in Fig. 4 consisting of two subnetworks N_1 and N_2 that perform the events e and f, respectively.

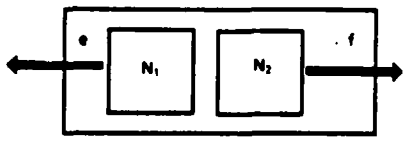


Figure 4. A network

The following cases show how $[N_1 \| N_2]$ depends on $[N_1]$ and $[N_2]$. We borrow notation from regular expressions. An infinite sequence of e's is denoted e^ω.

$[N_1]$	$[N_2]$	$[N_1 \| N_2]$
e^ω	f^ω	all fair merges of e^ω and f^ω.
e^ω	f^*	$(e \cup f)^* e^\omega$
e^ω	$\uparrow$	e^ω
e^k	$\uparrow$	$e^k \uparrow$

5. Specifications

In this section we present a way to use the model for specification and verification of networks.

The behavior of a network is specified through properties of its quiescent traces. These are stated as formulas in predicate logic. The quiescent traces should be specified indirectly by properties of their projections onto channels of the network. This is important when formulating proof rules for composition and abstraction. For instance, a specification of a subnetwork stating *"each quiescent trace only contains events on the channel c"* can not be used to specify quiescent traces of a larger network, but the specification *"the projection of each quiescent trace onto the channels of this subnetwork only contains events on the channel c"* can, since it will be true for any computation in which the subnetwork participates.

A specification language L therefore (following [ZRE]) contains the following special variables with intended interpretations:

π_{cset} : the projection of a quiescent trace onto the set of channels *cset*.

c : the sequence of messages on the channel c in a quiescent trace. Compare $\pi_{\{c\}}$ which denotes a sequence of events $<(c, m_1)(c, m_2) \ldots >$, and c which denotes a sequence of messages $< m_1 m_2 \ldots >$.

In addition L contains constants and variables of types sequence, message, integer, logical connectives and quantifiers. We also use the following functions and predicates.

$<>$	is the empty sequence.
ss'	is the concatenation of the sequences s and s'.
$s \leq s'$	states that the sequence s is a prefix of s'.
$s < s'$	states that the sequence s is a proper prefix of s'.
$s \preceq s'$	states that the sequence s is a (not necessarily consecutive) subsequence of s'.
$\lvert s \rvert$	is the (possibly infinite) length of the sequence s.

A specification S in the language L is a formula in which no special projection variable π_{cset} or c is bound by a quantifier. Define

chan(N) denotes the set of channels in N's signature.

chan(S) denotes the set of channels in the special variables π_{cset} and c that occur in S, i.e. the set of channels mentioned by S.

Definition 5.1. *A network N satisfies a specification S, written as*

$$N \text{ sat } S$$

iff the following holds:

1) each quiescent trace q of N satisfies the formula S. We assume that each special variable π_{cset} is interpreted as the projection of q onto the set of channels *cset*, and each special variable c is interpreted as the sequence of messages on the channel c in q.
2) *chan*$(S) \subseteq$ *chan*(N). The motivation is that a specification of N must not state properties of channels that are not channels of N. This has importance for the composition rule.
3) $[N]$ does not contain any divergent traces. The motivation is that we regard divergence as always undesirable.

□

Note that the specification method cannot describe networks that have divergent traces, in contrast to the model. The model describes what can be "observed" from the network; the specification method treats some observations as undesirable. The motivation for this discrepancy is to make the model flexible. It can be used for different specification methods with different considerations about desirable observations.

6. Proof System

6.1. Proof Rules

Theorem 6.1. *The following proof rules for composition, abstraction and renaming are sound.*

Composition: Composition of networks corresponds to conjunction of specifications.

$$\frac{N_i \quad \text{sat} \quad S_i \qquad i = 1, \ldots, k}{N_1 \| \cdots \| N_k \quad \text{sat} \quad \bigwedge_i S_i}$$

Abstraction: Let C be a set of internal channels of N. Let π_C and π_{NC} denote the projection of q onto the channels in C and onto the channels of N that are not in C, respectively.

$$\frac{\begin{array}{l} N \quad \text{sat} \quad S \\ S \ \wedge \ |\pi_{NC}| \neq \infty \quad \rightarrow \quad |\pi_C| \neq \infty \\ S \rightarrow S' \\ chan(S') \subseteq chan(N \setminus C) \end{array}}{N \setminus C \quad \text{sat} \quad S'}$$

Renaming: Let $S[\Phi]$ be the result of textually replacing all occurrences of channel names (e.g. in special variables) c by $\Phi(c)$ in S.

$$\frac{N \quad \text{sat} \quad S}{N[\Phi] \quad \text{sat} \quad S[\Phi]}$$

Consequence:

$$\frac{\begin{array}{ccc} N & \text{sat} & S \\ S & \rightarrow & S' \end{array}}{N \quad \text{sat} \quad S'}$$

□

6.2. Soundness

The soundness of the rules follows from theorem 4.3. Proof sketches for the composition and abstraction operations are given below.

Composition: The requirements on $N_1 \| \cdots \| N_k$ sat $\wedge_i S_i$ in definition 5.1 are motivated as follows:

1) From theorem 4.3 it follows that each projection of a quiescent trace q of $N_1 \| \cdots \| N_k$ onto a subnetwork N_i is a quiescent trace of N_i. Each projection onto a subnetwork N_i satisfies S_i. Since a specification only talks about q through its projections, the trace q must satisfy $\wedge_i S_i$.
2) The formula $\wedge_i S_i$ only mentions the channels of the subnetworks. These are also channels of $N_1 \| \cdots \| N_k$.
3) Theorem 4.3 shows that if no subnetwork has a divergent trace , then $N_1 \| \cdots \| N_k$ cannot have a divergent trace.

Abstraction: The requirements on $N \setminus C$ sat S' are motivated as follows:

1) Suppose q' is a quiescent trace of $N \setminus C$. Theorem 4.3 shows that q' is the projection of a quiescent trace q of N onto $chan(N \setminus C)$. Since q satisfies S and hence also S', q' must satisfy S'.
2) Follows from the fourth premise.
3) The first antecedent implies that N has no divergent traces. Theorem 4.3 then states that $N \setminus C$ has divergent traces iff there is an infinite trace of N whose projection onto the channels not in C is finite. The second premise states that this cannot happen.

6.3. Completeness

The rules are complete in a rather weak sense. To make this precise, first make the following definition

- A specification S is a *precise* specification of the network N iff $q \in [N]$ for each sequence q of events on the channels in $chan(N)$ that satisfies S (the special variables π_{cont} and c in S are interpreted as projections of q). In other words, the sequences of events in E_N that satisfy S are exactly those in $[N]$.

The rules are then complete in the following sense.

- Assume that N sat S for a network N that is defined from the subnetworks $N_1, N_2, \ldots, N_k$ using the composition, abstraction and renaming operations. Further assume that no part of the definition of N describes a network that has a divergent trace. If for each subnetwork N_i there is a specification S_i that makes the formula N_i sat S_i precise, then N sat S is provable from the precise specifications $S_1, \ldots, S_k$ of the subnetworks.

A necessary condition for completeness is thus that the specification language L can express precise specifications of the subnetworks that are considered.

However, the rules are incomplete in the following sense. Assume that a formula N sat S is derived from specifications N_i sat S_i of subnetworks, where some of the specifications are not precise. It may then be the case that N also satisfies a stronger specification S', but that this is not provable from the imprecise specifications.

The situation can be improved by introducing the following rule.

Entailment: Assume that every network that satisfies S also satisfies T.

$$\frac{N \quad \text{sat} \quad S}{N \quad \text{sat} \quad T}$$

The rule is a stronger version of the consequence rule, and takes into account properties true of all networks (such as those in proposition 4.2). When making proofs from imprecise specifications the rule is sometimes necessary.

7. Verification of the Alternating Bit Protocol

To illustrate the use of the proof system we specify and verify the alternating bit protocol. This is a simple protocol for transmitting messages correctly across a faulty medium.

The structure of the protocol is shown in Fig. 5. (notation from [Ha]).

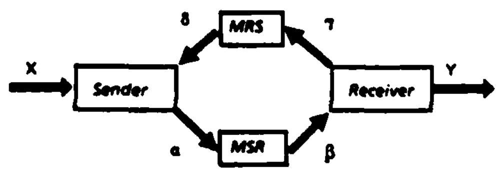


Figure 5. The alternating bit protocol

There are four modules: a *Sender*, a *Receiver* and the two media *MSR* and *MRS*. There are six channels: *X*, *Y*,

α, β, γ and δ.

The purpose of the protocol is to transmit all messages on the input channel X to the output channel Y in correct order, in spite of the fact that the media can lose messages.

The medium *MSR* is used to send messages from the *Sender* to the *Receiver*. Each message will also contain a sequence number, either 0 or 1. The medium *MRS* is used to send acknowledgments in the other direction. As acknowledgment for a message the same message is returned with its sequence number. Each medium can lose but not reorder messages. Corruption of messages can also be taken into account by modeling it as loss (some mechanism will detect and discard a corrupted message). Each medium is "fair" in the sense that if infinitely many messages are input, then infinitely many messages will be delivered.

The operation of the protocol is the following:

The *Sender* reads the pending message from X. It adds a sequence number to the message, transmits it on α and awaits an acknowledgment on δ. If it arrives, the procedure is repeated with the next message from X but with the sequence number inverted. If no acknowledgment arrives within a specified time period the sender retransmits the message. Retransmissions are repeated until an acknowledgment arrives.

The *Receiver* acknowledges each message received on β by sending it on γ. The first message on β and each subsequent message with a sequence number *different from* that of the previous one is delivered on Y.

We add the following notation:

$\mathcal{R}(s)$	reduces s by deleting all consecutive duplicates in the trace s. Thus $\mathcal{R}(m_1m_1m_1m_2m_1m_3m_3) = m_1m_2m_1m_3$.
number(s)	adds sequence numbers to the messages in s, alternatingly 0's and 1's, starting with 0.
unnumber (s)	is a sequence starting with the first message of s. It thereafter contains the sequence of messages in s that have a sequence number different from the previous one. The sequence numbers are deleted from the result.
X	is a special variable denoting the sequence of messages transmitted on X; similarly for $\alpha, \beta, \gamma, \delta$ and Y.

The following list of small lemmas will be used in the subsequent proof. Here s_1, s_2 denote sequences and m denotes a message.

$$s_1 \preceq s_2 \rightarrow \mathcal{R}(s_1) \preceq \mathcal{R}(s_2) \qquad (L1)$$

$$|s_1| < \infty \rightarrow (s_1 \preceq s_2 \wedge s_2 \preceq s_1 \rightarrow s_1 = s_2) \qquad (L2)$$

$$|s_1| < \infty \rightarrow \begin{pmatrix} (s_1 \preceq s_2) \wedge (s_2 \preceq s_1 < m >) \\ \rightarrow \\ (s_1 = s_2) \vee (s_1 < m >= s_2) \end{pmatrix} \qquad (L3)$$

$$unnumber(number(s)) = s \qquad (L4)$$

$$\mathcal{R}(s_1) \neq \mathcal{R}(s_2) < mm > \qquad (L5)$$

The proof also uses the following lemma implied by the entailment rule

$$\frac{N \quad \text{sat} \quad \begin{pmatrix} (|X| < \infty) \rightarrow (X = Y) \\ \wedge \ (|X| = \infty) \rightarrow (|Y| = \infty) \end{pmatrix}}{N \quad \text{sat} \quad X = Y} \qquad (L6)$$

which follows from proposition 4.2.

7.1. Specification of the Protocol

Specification of *MSR*.

$$\beta \preceq \alpha \qquad (MSR1)$$

$$|\alpha| = \infty \rightarrow |\beta| = \infty \qquad (MSR2)$$

The formula (MSR1) states that the medium may lose but not reorder messages. (MSR2) says that the medium is fair with respect to delivery of messages. If infinitely many messages are input, then infinitely many messages will be delivered.

The specification of *MRS* is similar.

$$\delta \preceq \gamma \qquad (MRS1)$$

$$|\gamma| = \infty \rightarrow |\delta| = \infty \qquad (MRS2)$$

Specification of *Sender*.

$$\mathcal{R}(\alpha) \preceq \mathcal{R}(\delta) \rightarrow \begin{bmatrix} \mathcal{R}(\alpha) = number(X) \\ \wedge \ |X| \neq \infty \rightarrow |\alpha| \neq \infty \end{bmatrix} \qquad (S1)$$

$$\mathcal{R}(\alpha) \not\preceq \mathcal{R}(\delta) \rightarrow \begin{bmatrix} (\exists m)[\mathcal{R}(\alpha) \preceq \mathcal{R}(\delta) < m >] \\ \wedge \ \mathcal{R}(\alpha) \preceq number(X) \\ \wedge \ |\alpha| = \infty \end{bmatrix} \qquad (S2)$$

Intuitively, the specification is motivated as follows. We assume that the sender discards messages on δ that are not the expected acknowledgment.

Suppose that q is a quiescent trace of the sender. According to the informal description of its operation, the sender reaches quiescence either by transmitting all messages of X on α and receiving acknowledgments, or by not receiving any acknowledgment for one message, which is retransmitted indefinitely.

In the former case the sequence of different messages on α is a subsequence of the different acknowledgments on δ (if extra acknowledgments on δ occur, they are discarded). Therefore the antecedent of (S1) is true. In this case, all messages of X are sent with sequence numbers on α (i.e. $\mathcal{R}(\alpha) = number(X)$), and the sender terminates after receiving the last acknowledgment (i.e. $|X| \neq \infty \rightarrow |\alpha| \neq \infty$) if X contains a finite number of messages.

In the latter case the antecedent of (S2) is true. Acknowledgments have arrived for all messages but the last (i.e. $(\exists m)[\ \mathcal{R}(\alpha) \preceq \mathcal{R}(\delta) < m >\ ]$) and the sender has only transmitted some of the messages from X (i.e. $\mathcal{R}(\alpha) \leq number(X)$). The last message on α is transmitted repeatedly (i.e. $|\alpha| = \infty$).

Specification of *Receiver*.

$$Y = unnumber(\mathcal{R}(\beta)) \qquad (R1)$$

$$\gamma = \beta \qquad (R2)$$

The formula (R1) states that the first message and subsequent ones with new sequence numbers received on β are transmitted onto Y. (R2) states that all messages are acknowledged.

7.2. Verification of the Protocol

We shall prove that the protocol behaves like a buffer. If all modules are composed and the internal channels α, β, γ and δ are abstracted, the resulting network satisfies the specification $X = Y$. The following specification will be proven

$$N \setminus \{\alpha, \beta, \gamma, \delta\} \quad \text{sat} \quad X = Y \qquad (*)$$

where N is the composition of the four modules

$$N = Sender \| MSR \| Receiver \| MRS$$

The formula $(*)$ is proven using the abstraction rule and the formula

$$N \text{ sat } \begin{bmatrix} X = Y \wedge \\ |X| < \infty \rightarrow |\alpha|, |\beta|, |\gamma|, |\delta| < \infty \end{bmatrix} \qquad (**)$$

The first conjunct $X = Y$ of $(**)$ follows by proving

$$N \quad \text{sat} \quad \begin{pmatrix} (|X| < \infty) \rightarrow (X = Y) \\ \wedge (|X| = \infty) \rightarrow (|Y| = \infty) \end{pmatrix} \qquad (***)$$

and using lemma (L6).

The formulas (MSR1), (MRS1) and (R2) together imply

$$\delta \preceq \gamma = \beta \preceq \alpha \qquad (1)$$

which by (L1) implies

$$\mathcal{R}(\delta) \preceq \mathcal{R}(\gamma) = \mathcal{R}(\beta) \preceq \mathcal{R}(\alpha) \qquad (2)$$

To prove the first conjunct of $(***)$ we now assume that $|X| < \infty$. Note that (S1) and (S2) imply $|\mathcal{R}(\alpha)| < \infty$. First assume that the antecedent of (S1) is true. This and (2) implies $\mathcal{R}(\delta) = \mathcal{R}(\alpha)$ using (L2), since $|\mathcal{R}(\alpha)|$ is finite. All relations in (2) must then (again using (L2)) be equalities, and we conclude $\mathcal{R}(\beta) = \mathcal{R}(\alpha)$.

The first conjunct of $(***)$ now follows from

$$\begin{aligned} Y &= unnumber(\mathcal{R}(\beta)) = unnumber(\mathcal{R}(\alpha)) = \\ &= unnumber(number(X)) = X \end{aligned} \qquad (3)$$

where the first equality follows from (R1), the second from the preceding paragraph, the third from (S1) and the last from (L4).

Next assume that the antecedent of (S1) is false, i.e. the antecedent of (S2) is true. We shall then derive a contradiction, showing that antecedent of (S1) must be true and (3) holds.

If the antecedent of (S2) is true, then the consequent of (S2) states that there is a message m such that $\mathcal{R}(\alpha) \preceq \mathcal{R}(\delta) < m >$. This together with (2) shows, using (L3), that either $\mathcal{R}(\delta) = \mathcal{R}(\alpha)$ or $\mathcal{R}(\delta) < m >= \mathcal{R}(\alpha)$. All but one of the relations in (2) must therefore be equalities, showing that either $\mathcal{R}(\beta) = \mathcal{R}(\alpha)$ or $\mathcal{R}(\beta) < m >= \mathcal{R}(\alpha)$. The following argument shows that the latter cannot be true:

Since $|\alpha| = \infty$ (from (S2)) and $|\mathcal{R}(\alpha)|$ is finite, α must be of the form $\alpha' m^\omega$ for some sequence α' whose last message is not m. Using (MSR1) and (MSR2) we get $\beta \preceq \alpha$ and $|\beta| = \infty$, which implies that β is of the form $\beta' m^\omega$ for some β' whose last message is not m. It follows that

$$\mathcal{R}(\alpha) = \mathcal{R}(\beta) < m >= \mathcal{R}(\beta' m^\omega) < m >= \mathcal{R}(\beta') < mm >$$

which is a contradiction since (L5) states that an expression of the form $\mathcal{R}(\alpha)$ can never contain two consecutive duplicates of the same message. Thus we must have $\mathcal{R}(\beta) = \mathcal{R}(\alpha)$.

The same argument can be used to show that $\mathcal{R}(\delta) = \mathcal{R}(\gamma)$ is true by excluding the case that there is a message m such that $\mathcal{R}(\delta) < m >= \mathcal{R}(\gamma)$.

Combining $\mathcal{R}(\beta) = \mathcal{R}(\alpha)$ and $\mathcal{R}(\delta) = \mathcal{R}(\gamma)$ in (2) yields $\mathcal{R}(\delta) = \mathcal{R}(\alpha)$. Thus the antecedent of (S2) is false, giving the desired contradiction.

To prove the second conjunct of $(***)$, assume that $|X| = \infty$. If $|\mathcal{R}(\alpha)| < \infty$ then the antecedent of (S2) must be true, since the consequent of (S1) states that $\mathcal{R}(\alpha) = number(X)$. But above it was shown that the antecedent of (S2) leads to a contradiction (note that we there never used the fact that $|X| < \infty$, only that $|\mathcal{R}(\alpha)| < \infty$). Thus $|\mathcal{R}(\alpha)| = \infty$. By (S1) and (S2) we get $\mathcal{R}(\alpha) \preceq \mathcal{R}(\delta)$. Using $\mathcal{R}(\alpha) = number(X)$ (from (S1)) and $\mathcal{R}(\delta) \preceq \mathcal{R}(\beta)$ (from (2)) we see that $\mathcal{R}(\beta)$ contains an infinite subsequence with alternating sequence numbers. Therefore $|unnumber(\mathcal{R}(\beta))|$ is infinite, whence $|Y| = \infty$ by (R1).

The proof of the first conjunct of $(**)$ is now completed. The second conjunct of $(**)$ is verified by assuming that $|X|$ and $|Y|$ are finite, and proving that the other channels also have finite length. This follows easily from the following formulas:

$$|X| \neq \infty \rightarrow |\alpha| \neq \infty \qquad (S1)$$
$$\beta \preceq \alpha \qquad (MSR1)$$
$$\gamma = \beta \qquad (R2)$$
$$\delta \preceq \gamma \qquad (MRS1)$$

Finally, (*) is proven by (**) and the abstraction rule.

Conclusion

We have presented a way to formalise the idea of quiescence for (nondeterministic) asynchronous networks ([Mis]). From an operational definition of networks we have derived a compositional model that represents both safety and liveness properties using quiescent traces.

A method for specification and verification of networks through properties of quiescent traces has been presented. We believe that the conciseness of the model allows short specifications and verifications, but more experience with examples is needed. It is sometimes difficult to prove properties of infinite computations. This was illustrated in the example, where it was easier first to carry out the proof assuming that the input sequence is finite, and then prove the infinite case by a special argument.

So far, no syntax for processes has been mentioned, from which a method of proving specifications about primitive networks (containing one process) could be constructed. We are experimenting with a small CSP-like language [H], but presently only a rudimentary finite-state language without variables or fairness constraints has been considered.

Acknowledgments

The author is grateful to Jay Misra for ideas and comments. His and K. Mani Chandy's ideas [Mis] provided the original inspiration for this work. Thanks to Ernst-Rüdiger Olderog for fruitful discussions, and in particular for the idea of using transition systems. Thanks also to Joachim Parrow for critical reading of the manuscript, and to Zohar Manna for fruitful comments on related work.

Appendix: Operational Semantics

Transition Systems

As described in section 3, a labeled transition system is a quintuple $< \Sigma, E, R, \sigma^0, \mathcal{F} >$. Here $\mathcal{F}$ is a finite collection of fairness sets [MP]. A fairness set is a set of transitions. An infinite sequence of transitions is *fair* iff for each fairness set $F \in \mathcal{F}$ the following holds

If the sequence contains infinitely many configurations in which a transition from F is enabled, then infinitely often a transition from F must occur.

As an example, channel fairness can be modeled by including, for each channel, a fairness set that contains all transitions corresponding events on this channel.

Operational Semantics

The basic unit of description is a subnetwork and not a process, as described in section 2. The transition system that models a primitive subnetwork, consisting of a process and its input channels, is obtained by composing a transition system that models the process, and transition systems that model the input buffers. This can be done using coupled transitions for communication, e.g. as in [Mil].

A network N is modeled by the labeled transition system $< \Sigma_N, E_N, R_N, \sigma_N^0, \mathcal{F}_N >$, which satisfies the following requirements that reflect the asynchronous nature of communication.

1) For each configuration $\sigma_N \in \Sigma_N$ and external input event $i \in E_N$ there is a transition $\sigma_N \xrightarrow{i} \sigma'_N \in R_N$ from σ_N labeled by i.
2) No fairness set in $\mathcal{F}_N$ may contain transitions labeled by external input events.
3) All transitions labeled by internal events, external output events, or silent events are elements of some fairness set.

The motivation for 1) and 2) is that a network has no control over external input events. The network can never refuse to receive input messages, and it can not constrain the occurrence of external input events in infinite transition sequences. Requirement 3) is a process liveness assumption.

Since communication events are atomic it is now possible to use the techniques in e.g. [BHR], [Mil] for synchronous networks, and model communication in a network by coupled transitions in subnetworks. The asynchronous nature of communication is captured in requirements 1) - 3). These requirements are sufficient to prove propositions 4.1 and 4.2 and theorem 4.3. It follows that our model can describe any type of network where components can be modeled by transition systems satisfying 1) - 3). In particular, it is not necessary that channels are perfect. The essential ingredient is that it is possible to send a message without cooperation by the receiver.

Operations

Composition, abstraction and renaming are defined as follows in the operational semantics.

Composition: Let $N = N_1 \| \ldots \| N_k$. The set of configurations Σ_N of N is the cartesian product $\Sigma_{N_1} \times \cdots \times \Sigma_{N_k}$ of the sets of configurations of the subnetworks. A configuration σ_N of N is a k-tuple, written

$$\sigma_N = \langle \sigma_{N_1}, \ldots, \sigma_{N_k} \rangle.$$

The transitions in R_N are obtained from the transitions of the subnetworks as follows: If e is an internal or silent event of a subnetwork N_i, or an external event of N, then transitions labeled by e only concern one component σ_{N_i}

of σ_N. This is formulated as the rule

$$\frac{\sigma_{N_i} \xrightarrow{e} \sigma'_{N_i}}{(\sigma_{N_1},\ldots,\sigma_{N_i},\ldots,\sigma_{N_n}) \xrightarrow{e} (\sigma_{N_1},\ldots,\sigma'_{N_i},\ldots,\sigma_{N_n})}$$

On the other hand, if e is an event on a channel internal to N, but external to two subnetworks N_i and N_j, then both components σ_{N_i} and σ_{N_j} are involved in a transition according to the rule

$$\frac{\sigma_{N_i} \xrightarrow{e} \sigma'_{N_i} \qquad \sigma_{N_j} \xrightarrow{e} \sigma'_{N_j}}{(\ldots,\sigma_{N_i},\ldots,\sigma_{N_j},\ldots) \xrightarrow{e} (\ldots,\sigma'_{N_i},\ldots,\sigma'_{N_j},\ldots)}$$

The network N inherits the fairness sets of its subnetworks: For each fairness set F_i of a subnetwork N_i there is a fairness set F of N that contains all transitions that are derived from a transition in F_i according to the above rules. Note that the transition system of N fulfills requirements 1) – 3) above.

Abstraction: For each configuration σ_N of N there is a configuration of $N \setminus C$, written $\sigma_N \setminus C$. Transitions labeled by events e not in C are unchanged

$$\frac{\sigma_N \xrightarrow{e} \sigma'_N}{\sigma_N \setminus C \xrightarrow{e} \sigma'_N \setminus C}$$

but events e on channels in C correspond to τ-transitions.

$$\frac{\sigma_N \xrightarrow{e} \sigma'_N}{\sigma_N \setminus C \xrightarrow{\tau} \sigma'_N \setminus C}$$

The network $N \setminus C$ inherits the fairness sets of N: for each fairness set F of N there is a fairness set of $N \setminus C$ containing all transitions derived from a transition in F according to the above rules.

Renaming: For each configuration σ_N of N there is a configuration of $N[\Phi]$, written $\sigma_N[\Phi]$. The transitions of $N[\Phi]$ are obtained by renaming the events of the transitions of N. Below, the function Φ denotes the extension of Φ from channel names to events (i.e. $\Phi((c,m)) = (\Phi(c),m)$ and $\Phi(\tau) = \tau$).

$$\frac{\sigma_N \xrightarrow{e} \sigma'_N}{\sigma_{N[\Phi]} \xrightarrow{\Phi(e)} \sigma'_{N[\Phi]}}$$

The network $N[\Phi]$ inherits the fairness sets of N, i.e. the fairness sets of $N[\Phi]$ are obtained by renaming the transitions in the fairness sets of N according to the rule above.

References

[BM] Back, R.J.R. and Mannila, H. "A refinement of Kahn's semantics to handle non-determinism and communication." *Proc. ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing*, 1982, pp. 111-120.

[BA] Brock, J.D. and Ackerman, W.B. "Scenarios: a model of non-determinate computation." In J. Dias, I. Ramos eds. *Formalisation of Programming Concepts, Lecture Notes in Computer Science 107*, Springer Verlag, 1981, pp. 252–259.

[Br] Broy, M. "Fixed point theory for communication and concurrency." In D. Bjoerner ed. *Formal Description of Programming Concepts II*, North Holland, Amsterdam, 1983, pp. 125-146.

[CH] Chen, Z.C. and Hoare, C.A.R. "Partial correctness of communicating processes and protocols." Technical monograph PRG-20, Programming Research Group, Oxford Computing Laboratory, May 1981.

[FLP] Frances, N., Lehmann, D., and Pnueli, A. "A linear history semantics for languages for distributed programming." *Theoretical Computer Science 32*, 1 (July 1984), pp. 25–46.

[Ha] Hailpern, B.T. "Verifying concurrent processes using temporal logic." *Lecture Notes in Computer Science 129*, Springer Verlag, 1982.

[Ho] Hoare, C.A.R. "Communicating sequential processes." *Comm. ACM 21*, 8 (Aug. 1978), pp. 666–676.

[K] Kahn, G. "The semantics of a simple language for parallel programming." *Proc. IFIP 74*, North-Holland, Amsterdam, 1974, pp. 471–475.

[MP] Manna, Z.M. and Pnueli, A. "How to cook a temporal proof system for your pet language." *Proc. 10th ACM Symposium on Principles of Programming Languages*, 1983, pp. 141-154.

[Mil] Milner, R. "A calculus of communicating systems." *Lecture Notes in Computer Science 92*, Springer Verlag, 1980.

[Mis] Misra, J. "Reasoning about networks of communicating processes." INRIA Advanced Nato Study Institute on Logics and Models for Verification and Specification of Concurrent Systems, Nice, France, 1984.

[MC] Misra, J. and Chandy, K.M. "Proofs of networks of processes." *IEEE Transactions on Software Engineering SE-7*, 4 (July 1981), pp. 417–426.

[MCS] Misra, J., Chandy, K.M,, and Smith, T. "Proving safety and liveness of communicating processes with examples." *Proc. ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing*, 1982, pp. 201-208.

[NGO] Nguyen, V., Gries, D., and Owicki, S. "A model and temporal proof system for networks of processes." *Proc. 12th ACM Symposium on Principles of Programming Languages*, 1985, pp. 121-131.

[Pa] Park, D. "The 'fairness' problem and nondeterministic computing networks." In de Bakker, Leuwen eds. *Foundations of Computer Science IV, Part 2*, Mathematical Centre Tracts 159, Amsterdam 1983, pp. 133–161.

[Pr] Pratt, V.R. "On the composition of processes." *Proc. 9th ACM Symposium on Principles of Programming Languages*, 1982, pp. 213–223.

[ZRE] Zwiers, J., de Roever, W.-P., and van Emde Boas, P. "Compositionality and concurrent networks: soundness and completeness of a proofsystem" Rep. 57, Informatica/Computer Graphics, Faculty of Science, Nijmegen University, The Netherlands, 1984. (to appear in ICALP 1985).

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER	2. GOVT ACCESSION NO.	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) Manuscript: A Novel Approach to Sequential Simulation		5. TYPE OF REPORT & PERIOD COVERED final: 6/14/81 - 6/15/85
		6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s) Devendra Kumar (Graduate Student) University of Texas at Austin		8. CONTRACT OR GRANT NUMBER(s) AFOSR 81-0205
9. PERFORMING ORGANIZATION NAME AND ADDRESS Computer Sciences Department University of Texas at Austin Austin, Texas 78712		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
11. CONTROLLING OFFICE NAME AND ADDRESS Capt. A. L. Bellamy AFOSR/NM Bolling AFB, DC 20332		12. REPORT DATE July 1985
		13. NUMBER OF PAGES
14. MONITORING AGENCY NAME & ADDRESS(if different from Controlling Office)		15. SECURITY CLASS. (of this report)
		15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

submitted to IEEE Transactions on Software and
IEEE Software II Conference

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

We present a novel approach to sequential simulation. In this approach we do not require events to be simulated in the chronological order of their occurrence. Instead, at any point in simulation all guaranteed events are simulated right away. This approach reduces the number of event list insertions in a number of simulation systems. In some cases it eliminates the need for event list algogether. It also reduces the number of scheduled events that do not take palce, i.e., get cancelled later in the simulation. Sometimes memory

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE

requirements may be reduced as well. We illustrate the approach with an example. The approach is based on a distributed simulation algorithm.

A Novel Approach To Sequential Simulation

Devendra Kumar

Department of Computer Sciences
University of Texas at Austin
Austin, TX 78712

ABSTRACT

We present a novel approach to sequential simulation. In this approach we do not require events to be simulated in the chronological order of their occurrence. Instead, at any point in simulation all guaranteed events are simulated right away. This approach reduces the number of event list insertions in a number of simulation systems. In some cases it eliminates the need for event list altogether. It also reduces the number of scheduled events that do not take place, i.e., get cancelled later in the simulation. Sometimes memory requirements may be reduced as well. We illustrate the approach with an example. The approach is based on a distributed simulation algorithm.

*This work was supported by Air Force Grant AFOSR 81-0205.

Table of Contents

1. Introduction

A fundamental convention in traditional simulation is that events are simulated in their chronological order of occurrence. The main reason behind this convention is as follows. An event simulated later but with an earlier time of occurrence may affect an event simulated earlier but with higher time of occurrence, making it incorrect. However, we observe that often in practice certain events can be guaranteed to be correct, and they are unaffected by the simulation of other events of earlier times of occurrence. We take advantage of this in our approach to simulation. We do not require events to be simulated in chronological order; rather, whenever an event is guaranteed to occur it is simulated right away.

The main advantage of this approach is that insertions of scheduled events (as in traditional simulation) can be reduced in number. One can often reduce the number of computations of scheduled events that would get cancelled later in traditional simulation. Sometimes this approach also results in reduced memory requirements.

The approach is derived from a distributed simulation algorithm. In distributed simulation, the simulator consists of a set of communicating processes which are assigned to several processors. We have adapted the algorithm to the case of a uniprocessor system. Due to the uniprocessor environment, several simplifications result, and new issues arise (e.g., scheduling of the processes). We have modified the algorithm accordingly.

2. The Approach

A system to be simulated consists of a set of *entities* that interact with each other. At discrete instants of time events occur – each event is caused by an entity and its occurrence affects the future behavior of zero or more entities. For example, an event in a communication network could be the sending of a message from one process to another; in this case the sender and the recipient processes are affected by this event. On the other hand, the event of broadcasting a message affects all the processes in the system.

We briefly review the traditional event driven simulation here. The simulation program maintains a list, called the *event list*, of scheduled events that might occur in the future. At an abstract level, a scheduled event can be defined by a tuple [e,t,i,S] where e is an identifier of the event, t is the time at which event e is supposed to occur, i is the entity that would cause e, and S is the set of entities that would be affected by it.

To simulate an event, the simulation program finds in the event list the scheduled event [e,t,i,S] with the minimum occurrence time t, and causes its entity i to simulate it, then advances the simulation clock to t. The entities in set S may be affected by the event occurrence – some of their old scheduled events may be deleted from the event list and new scheduled events may be inserted into it.

At a given point in simulation, a scheduled event [e,t,i,S] is said to be *guaranteed* if, based upon the events simulated so far, it is determined that this event will definitely be simulated. In other words, simulation of any other events before this scheduled event can not cancel it. As noted above, in general not all scheduled events are guaranteed. This results in the following fundamental convention in traditional simulation – events are simulated in the chronological order of their times of occurrence. In other words, an event is simulated only after all events of earlier times of occurrence have been simulated. (This convention is also followed in *time driven simulation*, and not just in *event driven simulation*, for the same reason.)

We observe, however, that often in practice many scheduled events are indeed guaranteed. For example, consider a FCFS queue with the convention that arrivals of input jobs are simulated in chronological order. Here the scheduled events of service completions are guaranteed to be simulated, since the arrival of new jobs can not cancel them. In our scheme we simulate such guaranteed events right away, instead of first depositing them into the event list and then waiting for the simulation clock to reach that time. More specifically, whenever an entity i computes an event e that is guaranteed to occur at time t, it goes ahead and simulates it. The tuple [e,t] is then deposited in a buffer $B_{i,j}$ for every other entity j affected by the event. An entity j computes its events based on the information it has received from its input buffers $B_{i,j}$, for various entities i.

Since events in the whole system are not being simulated in chronological order, there is no simulation clock maintained by the program. However, for any two entities i and j, all the events that are caused by i and affect j are simulated in chronological order. Thus, when a tuple [e,t] is deposited by i in the buffer $B_{i,j}$, entity j knows the entire history of all events that are caused by i and affect j up to time t. This helps j in its computation of future events.

Obviously, by computing guaranteed events and depositing them in buffers, we are avoiding the corresponding insertions in the event list, as required in traditional simulation.

The simulation program cycles through the entities – each entity computes guaranteed events and deposits them in the corresponding buffers. Each entity also discards from its input buffers those elements that are no longer needed for future computations.

What happens when no entity can compute a guaranteed event? In such a situation we revert back to the event list mechanism – the next scheduled events are computed and the event with the minimum occurrence time is simulated. Subsequently, guaranteed events are simulated till the above situation arises again. Thus in the total simulation, the simulator keeps alternating between "compute and simulate guaranteed events" and "compute and simulate the scheduled event with the minimum occurrence time" phase. (Henceforth we will call these phases: A and B, respectively.)

Advantages Of The Approach

In our approach we simulate guaranteed events right away. This avoids the corresponding insertions in the event list. Insertions in the event list may be quite time consuming, since the elements in the list need to be maintained in the order of increasing time values.

Consider a scheduled event in traditional simulation that gets cancelled later during the simulation. Obviously, the time involved in its computation and insertion in the event list goes wasted. This can happen, for example, in a priority queue where the arrival of a job of higher priority will preempt the current job in service. In our approach the number of such cases can be reduced. This stems from the fact that events need not be simulated in chronological order. For example, for a priority queue guaranteed output events may be computed by first simulating input events up to an appropriately higher time.

Sometimes our approach may result in reduced memory requirements. For example, in a tandem network of FCFS queues, in traditional simulation we need enough buffers to hold all the jobs present in the system at any time. In our approach, we may simulate the complete progress of one job, then simulate the complete progress of the next job, etc. Thus we would require memory to hold one job only.

3. An Example

We illustrate our simulation method by considering a simple system – a driver's license office, shown schematically in figure 1 below.

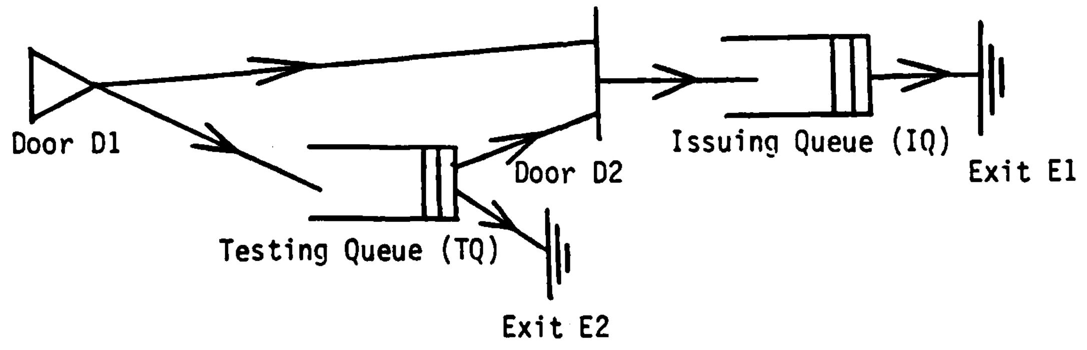


Figure 1: A Driver's License Office

Applicants for a license enter the office via door D1. There are two kinds of applicants – those who currently hold a license and simply want to renew it, and those who currently don't have a license and wish to get one. Applicants for a renewal enter a "license issuing area" via door D2. There is an issuing officer who takes their application, verifies the information therein, takes a photograph, accepts the license fee, and issues a license. Having received the license, the applicant leaves the driver's license office via exit E1.

Applicants without a license go to a "testing area" where a testing officer gives them a driving test. Some of them fail the test, and leave the testing area via exit E2. The successful applicants go to the license issuing area via door D2. From here on, they go through the same activities as described above for the license renewal applicants.

All the applicants arriving at the license issuing area form a waiting line (called the issuing queue or IQ). The issuing officer deals with one applicant at a time, in the FCFS order. Similarly, all applicants going to the testing area form a waiting line (called the testing queue or TQ), and the testing officer gives them the test in the FCFS order.

The problem is to simulate the events that occur in this system. An event is the arrival of an applicant at D2, IQ, TQ, E1 or E2.

In an actual simulation, one would normally compute the interarrival times at the door D1, and service times for the queues IQ and TQ by sampling from certain prespecified probability distributions. Similarly, one would determine the type of an applicant (whether applying for a renewal or otherwise) and whether an applicant taking the driving test fails, by sampling from prespecified probabilities. However, *for* *the ease of exposition*, we will assume a deterministic system with the following characteristics:

1. Applicants arrive at the door D1 at times 100, 200, 300, ... Service times at IQ and TQ are 105 and 500 respectively.

2. Applicants 10, 20, ... need to take the test; others are applying for a renewal.

3. For the applicants taking the test – the first one fails the test, next one passes, third one fails, and so on.

4. We assume that only 20 applicants enter the system.

We assume that the only information of interest about an applicant is his id. Hence the element e in a tuple [e,t] would refer to the applicant's id. We will use the following order in which the entities are reached by the simulation program to compute their guaranteed events. The phase of computing guaranteed events (i.e., phase A) consists of an alternating sequence of two subphases. In subphase 1, we follow the progress of applicants from the door D1 to exit E2 or door D2 (as the case may be), one applicant at a time. This subphase is over when an applicant reaches door D2. In subphase 2, we follow the progress of applicants from the door D2 to exit E1, one applicant at a time. This subphase is over (and subphase 1 starts) when no more applicants can progress from door D2. At the start of subphase 1, if no applicants can progress, then we enter phase B of the algorithm. In phase B, obviously we have to consider only the scheduled event for the door D2.

Below we show the sequence of actions taken by the simulation program. Specifically, we show the sequence in which tuples are computed for various buffers, and the computation of scheduled events in phase B. In the following, a quadruple (e,t,i,j) is used to state that tuple [e,t] is deposited in the buffer $B_{i,j}$. We refer to an entity i by its symbolic name D1, D2, E1, E2, IQ, or TQ, instead of an integer. The buffers are referred to in the similar way. Applicant ids are assumed to be A1, A2, ...

1. **Tuples buffered:**

   (A1,100,D1,D2), (A2,200,D1,D2), ..., (A9,900,D1,D2).

   This simulates the arrivals of applicants A1, ..., A9 at door D2 at times 100, ..., 900. During this simulation period, entity D2 can not compute its guaranteed next output events, since it has to output the tuples in buffer (D2,IQ) in the chronological order. If it deposits the tuple (A1,100), it doesn't know if there would be a tuple deposited on buffer (TQ,D2) later with a time component less than 100.

2. **Tuples buffered:**

   (A10,1000,D1,TQ), (A10,1500,TQ,E2).

   This simulates the arrivals of applicant A10 at TQ and E2 at times 1000 and 1500 respectively.

3. **Tuples buffered:**

   (A11,1100,D1,D2), (A12,1200,D1,D2), ..., (A19,1900,D1,D2).

This simulates the arrival of applicants 11, ..., 19 at door D2 at times 1100, ..., 1900. Note that entity D2 still cannot compute an output tuple.

4. **Tuples buffered:**

(A20,,2000,D1,TQ), (A20,2500,TQ,D2)

This simulates the arrivals of applicant A20 at TQ and D2 at times 2000 and 2500 respectively.

5. **Tuples buffered:**

(A1,100,D2,IQ), (A1,205,IQ,E1), (A2,200,D2,IQ), (A2,310,IQ,E1), ..., (A9,900,D2,IQ), (A9,1045,IQ,E1), (A11,1100,D2,IQ), (A11,1205,IQ,E1), (A12,1200,D2,IQ), (A12,1310,IQ,E1), ..., (A19,1900,D2,IQ), (A19,2045,IQ,E1).

This simulates the arrivals of applicants A1, ..., A9 and A11, ..., A19 at IQ and E1. Note that after computing the above tuples, D2 can not compute its next output tuple since it doesn't know the simulation time of next tuple to arrive in the buffer (D1,D2). At this point D1 also can not compute an output (since it has simulated all 20 applicants). Hence the algorithm enters its phase B. As mentioned before, we need compute only the next scheduled event for D2. This scheduled event is the tuple (A20,2500).

6. Tuples buffered:

(A20,2500,D2,IQ), (A20,2605,IQ,E1).

This simulates the arrivals of applicant A20 at IQ and E1. At this point the algorithm again enters phase B. Since there is no scheduled event for D2, simulation ends.

4. Further Improvements

We mentioned in section 2 that the simulation program cycles through various entities to compute their guaranteed events. It is possible that when a particular entity is reached, it has nothing to output. This wastes the time involved in reaching this entity and checking this condition for the entity. Depending on the particular system being simulated, one could possibly define an order in which to reach the entities such that the number of such cases is reduced. This order could be defined either statically or dynamically. For example, in a tandem network of FCFS queues, one may use the same order of entities in which a job arrives at them (statically defined order). In simulating a tree network rooted at a source process that generates jobs, one may follow the progress of one job from the source to a sink, then follow the next job, etc. (dynamically defined order). Now we define a heuristic to reduce the number of such cases in general. We keep a list of "potentially active" entities. Any entity currently not on this list is guaranteed not to be able to compute a guaranteed event. The simulation program reaches the entities by going through this list. When an entity has computed all its guaranteed events, it is removed from the list. When does an entity enter the list? One heuristic would be – whenever it receives an input. In a specific application, one could possibly define more appropriate boolean conditions for the specific entities. It would be helpful to keep a boolean variable for each entity to check whether it is on the list currently; it should be checked before evaluating the above boolean condition.

Consider an instant when phase A is over, i.e., no guaranteed events can be computed for any entity. Which entities should compute their next events? In general not all of them. For a particular system certain entities may be known not to compute the scheduled event with minimum time. A FCFS queue is one such example. We need not consider such entities in computing the scheduled events.

Earlier we suggested that in phase B we compute all the scheduled events afresh, i.e., the scheduled events computed in the current occurrence of phase B are not saved to be used in the next phase. Sometimes, several scheduled events computed in phase B remain valid even in the next occurrence of phase B. Here we suggest a heuristic to take advantage of this. One may keep an event list of the scheduled events computed during phase B. One would also keep a list of those entities whose scheduled events must be computed at the next occurrence of phase B. (If these entities have elements in the event list then they must be removed from the event list in the next occurrence of phase B.) This list is similar to the "potentially active" list mentioned above. As before, appropriate conditions may be defined to decide when an entity should be inserted in this list. Also, a boolean variable may be kept for each entity to check if it is currently on this list.

5. Discussion and Conclusions

We have presented a new approach to sequential simulation. In this approach we do not require that events are simulated in their chronological order. This is a major point of deviation from traditional simulation. This approach results in reduced number of insertions in the event list. It can sometimes reduce the number of computations of scheduled events that do not actually take place. Also, it can reduce memory requirements. These advantages would depend on the specific system being simulated. For better performance, one has to take decisions regarding the following issues: (i) In what order are the entities reached to compute their guaranteed events, (ii) When phase B begins, which entities should be reached to compute their scheduled events, and (iii) Should one keep an event list to hold scheduled events previously computed, so that some of them could be used in the next occurrence of phase B. We have suggested heuristics for these issues.

Our approach is based on a distributed simulation algorithm. In distributed simulation, usually each entity is simulated by an autonomous process and various processes are mapped onto processors. In order to achieve a high degree of parallelism, the processes are asynchronous and there is no global simulation clock. Processes synchronize with each other by sending and receiving the tuples [e,t]. In general, deadlocks may arise resulting from a cyclic waiting among the processes. One method of handling the deadlock problem is to let the simulator deadlock, to detect deadlock, and to recover from it [Chandy 81, Kumar 85c]. We applied this algorithm to the case of a uniprocessor system. In such a system, deadlock is easy to detect (when no entity can compute a guaranteed event, i.e., the list of "potentially active" entities is empty).

Several other algorithms have been proposed for distributed simulation [Chandy 79a, Peacock79, Jefferson82, Misra 84, Kumar 85c], but they involve too many overhead messages and we expect that this may severely degrade performance on a uniprocessor system.

We have shown in this paper that an algorithm developed for distributed simulation could also be useful for sequential simulation and it can suggest a new approach in this environment. We expect similar lessons to be learnt in other application areas of parallel computing.

Acknowledgements

My special thanks to Professor M. Gouda for his help in the presentation of this work, including several fruitful discussions and comments on earlier drafts of the paper. I am also thankful to Professors J. Misra and K. M. Chandy for several discussions, encouragement, and financial support for this work.

References

[Bagrodia 83]	R. Bagrodia, "May: A Process Based Simulation Language", Master's Report, Dept. of Computer Sciences, University of Texas at Austin, Austin, Texas 78712, 1983.
[Birtwistle 73]	G. M. Birtwistle, O. J. Dahl, B. Myhrhaug, and K. Nygaard, "Simula Begin", Auerbach Publishers Inc., Philadelphia, Pennsylvania, 1973.
[Birtwistle 79]	G. M. Birtwistle, "DEMOS: A System For Discrete Event Simulation", Macmillan Press, 1979.
[Bryant 77]	R. E. Bryant, "Simulation of Packet Communication Architecture Computer Systems", Technical Report MIT, LCS, TR-188, Massachusetts Institute of Technology, November 1977.
[Chandy 79a]	K. M. Chandy, V. Holmes, and J. Misra, "Distributed Simulation Of Networks", *Computer Networks* 3(1):105-113, Feb. 1979.
[Chandy 79b]	K. M. Chandy and J. Misra, "Distributed Simulation: A Case Study In Design And Verification of Distributed Programs", *IEEE Transactions on Software Engg.*, SE-5(5):440-452, September 1979.
[Chandy 81]	K. M. Chandy and J. Misra, "Asynchronous Distributed Simulation Via a Sequence of Parallel Computations", *Communications of the ACM*, Vol. 24, No. 4, pp.198-205, April 1981.

[Chandy 82] K. M. Chandy and J. Misra, "A Distributed Algorithm for Detecting Resource Deadlocks in Distributed Systems", *ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing*, Ottawa, Canada, August 1982.

[Chandy 83] K. M. Chandy, J. Misra, and L. Haas, "Distributed Deadlock Detection", *ACM Transactions on Computing Systems*, Vol. 1, No. 2, pp. 144-156, May 1983.

[Christopher 83] T. Christopher, et. al., "Structure of a Distributed Simulation System", in *Proceedings of the 3rd International Conference on Distributed Systems*, Ft. Lauderdale, Florida, 1983.

[Dahl 70] O. J. Dahl, B. Myhrhaug, and K. Nygaard, "Simula 67 Common Base Language", Norwegian Computing Centre, Oslo, Norway, 1970.

[Fishman 78] G. S. Fishman, "Principles of Discrete Event Simulation", A Wiley-Interscience Publication, John Wiley and Sons, New York, New York, 1978.

[Franta 77] W. R. Franta, "Process View of Simulation", Elsevier Computer Science Library, Operating and Programming Systems Series, P.J. Denning (ed.), Elsevier North Holland Publisher, 1977.

[Gligor 80] V. Gligor and S. Shattuck, "On Deadlock Detection in Distributed Data Bases", *IEEE-TSE*, Vol. SE-6, No. 5, September 1980.

[Holt 72] R. C. Holt, "Some Deadlock Properties of Computer Systems", *Computing Surveys*, Vol. 4, No. 3, pp. 179-196, September 1972.

[Jefferson 82] D. R. Jefferson and H. A. Sowizral, "Fast Concurrent Simulation Using The Time Warp Mechanism, Part I: Local Control", Technical Report, The Rand Corporation, Santa Monica, California, December 1982.

[Kleinrock 76] L. Kleinrock, "Queueing Systems, Volume II: Computer Applications", Wiley-Interscience, John Wiley & Sons, Inc., 605 Third Avenue, New York, N.Y. 10158, 1976.

[Kobayashi 81] H. Kobayashi, "Modeling and Analysis: An Introduction to Systems Performance Evaluation Metodology", The Systems Programming Series, Addison-Wesley Publishing Company, Menlo Park, California, Oct. 1981.

[Kumar 85a] D. Kumar, "A Class of Termination Detection Algorithms For Distributed Computations", Technical Report, Department of Computer Sciences, University of Texas at Austin, Austin, Texas 78712, May 1985.

[Kumar 85b] D. Kumar, "A High Speed Distributed Simulation Scheme And Its Performance Evaluation", in preparation.

[Kumar 85c] D. Kumar, "Distributed Simulation", Ph.D. Thesis (in preparation), Department of Computer Sciences, University of Texas, Austin, Texas 78712.

[Lamport 78] L. Lamport, "Time, Clocks, and the Ordering of Events in a Distributed System", *Communications of the ACM,* Vol. 21, No. 7, July 1978.

[Lonow 82] G. Lonow and B. Unger, "Process View of Simulation In ADA", in *1982 Winter Simulation Conference*, pages 77-86, 1982.

[Misra 83] J. Misra, "Detecting Termination of Distributed Computations Using Markers", *Proceedings of the ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing*, Montreal Canada, August 17-19, 1983.

[Misra 84] J. Misra, "Distributed Simulation", *IEEE Tutorial on Distributed Simulation*, 1984.

[Obermarck 82] R. Obermarck, "Distributed Deadlock Detection Algorithm", *ACM Transactions on Database Systems*, Vol. 7, No. 2, pp.187-208, June 1982.

[Peacock 79] J. K. Peacock, J. W. Wong, and E. G. Manning, "Distributed Simulation Using A Network of Processors", *Computer Networks* 3(1):44-56, Feb. 1979.

[Sauer 78] C. H. Sauer, "Characterization And Simulation of Generalized

Queuing Networks", Research Report RC-6057, IBM Research, Yorktown Heights, NY, May 1978.

[Sauer 81] C. H. Sauer, and K. M. Chandy, "Computer Systems Performance Modeling", Prentice-Hall, Inc., Englewood Cliffs, New Jersey 07632, 1981.

[Sauer 83] C. H. Sauer and E. A. MacNair, "Simulation of Computer Communication Systems", Prentice-Hall, Inc., Englewood Cliffs, New Jersey 07632, 1983.

[Seethalakshmi 79] M. Seethalakshmi, "A Study And Analysis of Performance of Distributed Simulation", Master's Report, Dept. of Computer Sciences, University of Texas, Austin, Texas 78712, May 1979.

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER	2. GOVT ACCESSION NO.	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) Manuscript: A High Speed Distributed Simulation Scheme and Its Performance Evaluation		5. TYPE OF REPORT & PERIOD COVERED final: 6/14/81 - 6/15/85
		6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s) Devendra Kumar (Graduate Student) University of Texas at Austin		8. CONTRACT OR GRANT NUMBER(s) AFOSR 81-0205
9. PERFORMING ORGANIZATION NAME AND ADDRESS Computer Sciences Department University of Texas at Austin Austin, Texas 78712		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
11. CONTROLLING OFFICE NAME AND ADDRESS Capt. A. L. Bellamy AFOSR/NM Bolling AFB, DC 20332		12. REPORT DATE July 1985
		13. NUMBER OF PAGES
14. MONITORING AGENCY NAME & ADDRESS(if different from Controlling Office)		15. SECURITY CLASS. (of this report)
		15a. DECLASSIFICATION/DOWNGRADING SCHEDULE
16. DISTRIBUTION STATEMENT (of this Report)		
17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)		
18. SUPPLEMENTARY NOTES submitted to 1) Nineteenth Annual Simulation Symposium 2) CMG '85		
19. KEY WORDS (Continue on reverse side if necessary and identify by block number)		

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

We present a high speed distributed simulation scheme that can be used to simulate any feedforward network of processes which communicate solely by exchanging messages. The scheme is simple to implement and the number of overhead messages is nearly zero. We prove the correctness of the scheme and study its performance both analytically and empirically. Under reasonable assumptions, it is shown that the scheme offers a substantial speed up over sequential simulation. In particular we show that for a large class of networks, the speed up over sequential simulation is proportional to N, where N is the number of processors used in the distributed simulator.

A High Speed Distributed Simulation Scheme And Its Performance Evaluation*

Devendra Kumar

Department of Computer Sciences
University of Texas at Austin
Austin, TX 78712

ABSTRACT

We present a high speed distributed simulation scheme that can be used to simulate any feedforward network of processes which communicate solely by exchanging messages. The scheme is simple to implement and the number of overhead messages is nearly zero. We prove the correctness of the scheme and study its performance both analytically and empirically. Under reasonable assumptions, it is shown that the scheme offers a substantial speed up over sequential simulation. In particular we show that for a large class of networks, the speed up over sequential simulation is proportional to N, where N is the number of processors used in the distributed simulator.

*This work was supported by Air Force Grant AFOSR 81-0205.

Table of Contents

1. Introduction

In general, simulation is one of the most expensive software to run; each simulation run usually requires large execution time and storage. The problem is compounded with the emerging need to simulate distributed systems that consist of large numbers of interacting components. One technique to counter this problem is to partition the simulation software into a number of autonomous processes that communicate solely by exchanging messages. These processes are executed in parallel on the different processors of a distributed system, thus reducing the total execution time. This technique is known as *distributed simulation* [Chandy 79a, Chandy 81], and the set of communicating processes that perform the simulation is called a *distributed simulator* (or *logical system*).

Many distributed simulation schemes have been proposed in the literature [Chandy 79a, Chandy 79b, Peacock 79, Chandy 81, Jefferson 83a, Jefferson 83b]. They can be classified into two categories depending on how they deal with deadlock situations that may arise between the communicating processes in a distributed simulator. The scheme in [Chandy 81] allows deadlocks to occur, but they are later detected and recovered from. The schemes in [Chandy 79a, Chandy 79b, Peacock 79] allow the processes to exchange overhead NULL messages whose sole purpose is to avoid communication deadlocks. For a detailed and complete survey on distributed simulation schemes, we refer the reader to [Misra 84, Kumar 85].

Unfortunately, except for one simulation study [Seethalakshmi 79] whose results are nonconclusive the performance of these existing schemes have not been analyzed yet. This situation leaves the interesting question "how good is distributed simulation versus sequential simulation?" basically unanswered. This paper represents a first step towards answering this question. In particular, we present a new distributed simulation scheme called TBASIC, and show that in many cases TBASIC can achieve a speed-up proportional to N over sequential simulation, where N is the number of processors used to run the distributed simulator.

Following the introduction, the paper is organized as follows. In Section 2, we present the scheme TBASIC and prove its correctness. In Section 3, we analyze the performance of TBASIC in simulating tandem queuing networks. In Section 4 we present an approximate analysis for the performance of TBASIC in simulating feedforward queuing networks, and report on an empirical validation of this approximate analysis. In Section 5, we use this approximate analysis to evaluate TBASIC, i.e., determine its performance in simulating various classes of feedforward queuing networks. In particular, we show that for a general class of these networks the speed up attained by TBASIC over sequential simulation is proportional to the number of processors in the distributed simulator. Concluding remarks are given in Section 6.

2. The Scheme TBASIC

In this section we characterize the physical systems that can be simulated using our scheme TBASIC. We then present TBASIC and discuss its correctness proof.

2.1 Physical Systems

We call the systems to be simulated by TBASIC physical systems. A *physical system* consists of a finite number of *physical processes* (or *pp's* for short) that interact with each other solely by exchanging messages via unbounded, one-to-one, one directional communication lines. The topology of a physical system is assumed to be acyclic, and its communication delays (i.e., the time between one process sending a message and the message being placed at an input line of its destination process) are assumed to be zero. [Kumar 85] discusses how any discrete event system can be modeled as a physical system.

The *message history* up to time t for a line (i, j) in the physical system is defined as the tuple sequence $<(t_1, m_1), (t_2, m_2), \dots (t_r, m_r)>$ where $m_1, m_2, \dots$ is the sequence of messages sent on this line up to time t, and $t_1 < t_2 < \dots < t_r \leq t$ are the times at which these messages were sent.

We assume that if the message histories of all input lines of a pp is known up to time t, then the message history on each output line of the same pp is computable up to at least time t. This is called the *realizability property.*

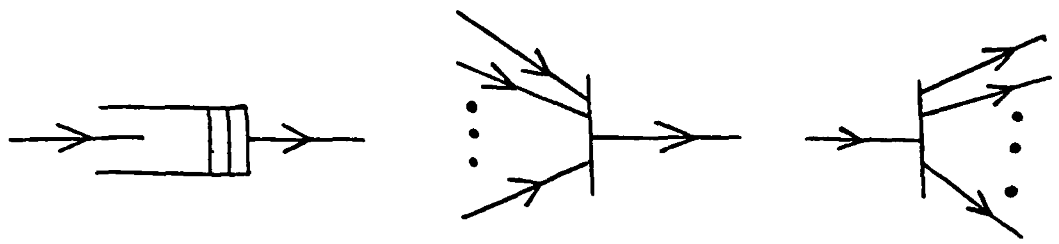
We assume that any physical system is required to be simulated for a time period [0, Z], where Z is any positive value.

Five example of pp's are as follows: (These are called *queuing processes* since as discussed later they can be used to model queuing networks. These processes are shown in figure 1.)

1. A *delay process* has one input and one output lines. It processes its input messages in a FCFS queuing discipline, then sends each of them out after a finite service time. The service time may be deterministic or probabilistic.

2. A *merge process* has two or more input lines, and one output line. Whenever it receives a message on one of its input lines, it sends it via its output line after zero delay. If occasionally two or more inputs are received at the same time, they are put together in a single message and sent out via the output line.

3. A *fork process* has one input line and two or more output lines. Whenever it



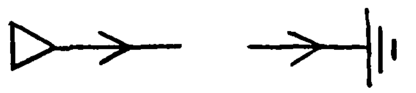


Figure 1: Queueing processes

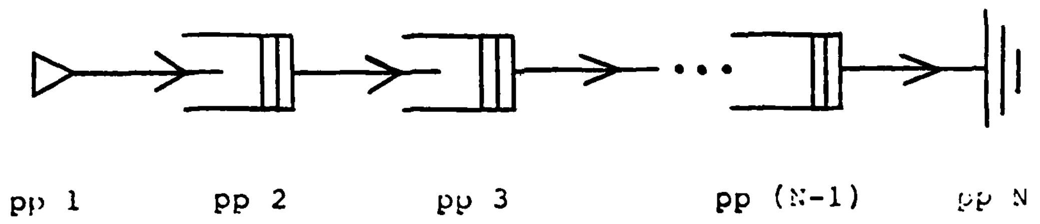


Figure 2: Tandem network of N processes

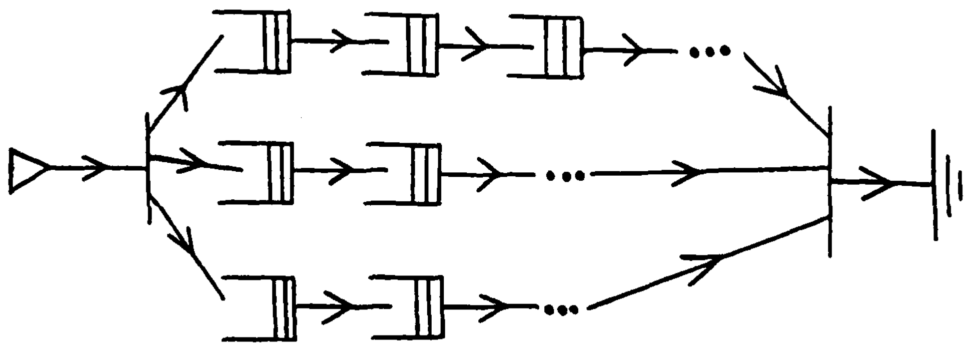


Figure 3: Parallel network

receives a message on its input line, it sends it out along one of the output lines, after zero delay. The output line for a message is chosen in a probabilistic manner, according to predefined branching probabilities for the output lines.

4. A *source process* has no input lines, and one output line. It simply generates messages and sends them out.

5. A *sink process* has one input line and no output lines. It simply absorbs its input messages.

2.2 TBASIC

In TBASIC, each physical system is simulated by a distributed simulator called a logical system. A *logical system* is a network of processes, called *logical processes* (or *lps* for short) that is topologically isomorphic to the physical system it simulates, with each pp being replaced and simulated by one lp. Each lp executes a loop consisting of three phases, called the computation phase, the termination checking phase, and the input phase.

1. In the *computation phase*, the lp computes for each output line a (possibly empty) set of output tuples (t, m) in the chronological order of their t components, and sends them out.

2. In the *termination checking phase* it checks if it has received an input with t-value $=$ Z on each input line. If so, it terminates. Otherwise it goes on to the input phase.

3. In the *input phase* it waits until at least one more message has arrived on its input port; it then receives all available messages in FIFO order and stores them in its line buffers, before returning to the computation phase.

In the computation phase, if the output history is known up to a time $\geq$ Z, then only the tuples, whose t-values were $\leq$ Z, are sent out. Then, if the last sent tuple on a line had a t-value $<$ Z, then a special *termination message* (Z, NULL) is sent out on that line. This tuple informs the receiving lp that no more tuples would be sent out on this line.

2.3 Correctness of TBASIC

We now show that TBASIC is both safe and live. The safety of TBASIC means that at any point in the simulation, the sequence of tuples sent or received on a line correctly simulates a sequence of messages sent on the corresponding line in the physical system. In other words, tuples are sent on every line in chronological order of their t-values, and every tuple (t, m), except for the termination message, corresponds to a message m sent on the corresponding line in the physical system at time t. Moreover, no message on that line is skipped in the sequence of tuples up to time t.

The liveness of TBASIC means that within a finite time from the start of simulation, a tuple with t-value = Z would be sent on each line in the logical system, and each lp would terminate. These safety and liveness properties follow from the next two theorems respectively.

Theorem 1: Let the sequence of sent tuples on some line (i, j) up to some point of the simulation be $<(t_1, m_1), ..., (t_k, m_k)>$. Then each of the following assertions holds.

1. $t_1 < t_2 < ... < t_k \leq Z$

2. For any tuple (t_s, m_s) in this sequence, except for the termination message if any, message m_s was sent on line (i, j) in the physical system at time t_s.

3. If a message m was sent in the physical system on the line (i, j) at time t where $t \leq t_k$, then the tuple (t, m) is present in the above sequence.

$\square$

A proof of this theorem is by induction on the number of events in the logical system; it is similar to that of theorem 4 in [Chandy 79a]. A proof of the next theorem is given in appendix A.

Theorem 2: A tuple with t-value = Z will be sent, and received, on every line in the logical system within a finite time from the start of the simulation.

$\square$

3. A Performance Analysis of TBASIC in Simulating Tandem Queuing Networks

In this section, we consider a class of physical systems, called tandem networks, and derive the ratio of the required simulation time when using a sequential simulator to simulate any network in this class to the required simulation time when using TBASIC to simulate the same network. More specifically, we derive a formula for the following ratio when the physical system is a tandem network:

SR = Speed-up ratio

$$= \text{SST/DST}, \qquad (1)$$

where SST = Sequential simulation time, i.e., the time taken by a sequential simulator to simulate some physical system up to time Z, and

DST = Distributed simulation time, i.e., the time taken by a logical system to simulate the same physical system up to Z.

A *tandem network* is a linear sequence of pps; the first pp in the sequence is a source, the last one is a sink, and the intermediate ones are delay pps (see figure 2). In TBASIC, a tandem network is simulated by a *logical system* that is a linear sequence of lps. The following parameters are used in the analysis below.

$N =$ the total number of pps in the tandem network
(It is also the number of lps in the simulating logical system.)

$M_{i,j} =$ the total number of messages sent out on line (i, j) in the physical system.
(It is also the total number of messages sent out on line (i, j) in the logical system.)

$1/\mu_i =$ the constant processing time taken by lp i in computing one output tuple

$\pi_{i,j} =$ the propagation delay on line (i, j), i.e., the time delay between the sending of a message by lp i on line (i, j) and its reception at the input port of lp j

$D_{i,j,r} =$ the departure time of the rth message on line (i, j) from lp i

$A_{i,j,r} =$ the arrival time of the rth message on line (i, j) to lp j

$D_{i,r} =$ the departure time of the rth message from lp i

$A_{i,r} =$ the arrival time of the rth message to lp i

A proof of the following theorem is in appendix A.

Theorem 3: For i = 1, 2, ..., N-1, and r = 1, 2, ...,

$$D_{i,r} = (\sigma_i + \phi_i - 1/\nu_i) + r/\nu_i \tag{2}$$

$$\text{where } \sigma_i = \sum_{k=1}^{i} (1/\mu_k)$$

$$\phi_i = \sum_{k=2}^{i} \pi_{k-1,k} \text{ and}$$

$$\nu_i = \min_{k=1}^{i} \{\mu_k\}$$

$\square$

From theorem 3,

$$A_{i,r} = D_{i-1,r} + \pi_{i-1,i}$$

$$= (\sigma_{i-1} + \phi_i - 1/\nu_{i-1}) + r/\nu_{i-1} \tag{3}$$

Therefore, the distributed simulation time DST can be computed as follows:

$$DST = \max_{i=2}^{N} \{A_{i,r}, \text{ where } r = M_{i-1,i}\}$$

$$= \max_{i=2}^{N} \{(\sigma_{i-1} + \phi_i - 1/\nu_{i-1}) + M_{i-1,i}/\nu_{i-1}\} \tag{4}$$

Formula (4) is exact; but to gain more insight into it, we better approximate it by the following (approximate) assumptions:

1. If the value of Z is sufficiently large then $M_{i-1,i}$ would be large, and DST can be written as:

$$DST \approx \max_{i=1}^{N-1} \{M_{i,i+1}/\nu_i\} \tag{5}$$

2. If all $M_{i,i+1}$'s are roughly equal (this would happen, for example, if the sum of service times of delay pps is smaller than interarrival times of messages generated by the source pp. This would also happen if the source pp stops producing messages long before time Z, so that the last message does reach the sink by time Z), say M, then DST can be written as

$$DST \approx M \cdot \max_{i=1}^{N-1} \{1/\nu_i\}$$

$$= M/\nu_{N-1} \qquad (6)$$

3. Furthermore, if all μ_i's were equal, say μ, then

$$DST \approx M/\mu \qquad (7)$$

The sequential simulation time SST may be approximated by

$$SST \approx \sum_{i=1}^{N-1} \{M_{i,i+1}/\mu_i\} \qquad (8)$$

This approximation is arrived at by several approximating assumptions. (For example, we have ignored the processing time involved in managing the event-list.) It is expected that, normally, the actual value of SST would be larger than the above approximation.

The speed up ratio SR can now be computed from (7) and (8).

$$SR = N - 1 \qquad (9)$$

This formula shows that the speed up offered by TBASIC in the case of tandem networks is proportional to N, where N is the number of processes in the distributed simulator.

4. An Approximate Performance Analysis of TBASIC in Simulating Feedforward Queuing Networks

An important class of physical systems that can be simulated using TBASIC is feedforward queuing networks. In section 4.1 we present an approximate analysis to compute the ratio SR when the simulated physical system is a feedforward queuing network that consists of pps from the five classes- delay, fork, merge, source, and sink defined in section 2.1. In section 4.2 we use simulation to validate this approximate analysis.

4.1 Approximate Analysis

In order to compute SR, we first compute the quantities $D_{i,j,r}$ for every line (i, j) in the logical system and for every value of r. (Recall that $D_{i,j,r}$ is the time at which the r^th^ message is sent along line (i, j) in the logical system.) The analysis is approximate, since it is based on the following three assumptions.

1. If $D_{i,j,r} = A + B * r$ where A and B are constants that do not depend on r, then $D_{i,j,r}$ can be approximated by $B * r$. Note that as simulation progresses

and r becomes large the effect of the constant term A diminishes. Similar approximation is made for $D_{i,r}$ also. (Recall that $D_{i,r}$ = the time at which lp i sends out its r^{th} message.)

2. For any fork lp i with one input line and n output lines $(i, j_1), \ldots, (i, j_n)$, and with branching probabilities $b_{i,j1}, \ldots, b_{i,jn}$ respectively,

$$D_{i,js,r} = D_{i,r}/b_{i,js} \text{ for every } s = 1, 2, \ldots, n.$$

3. The message flow on every line in the physical system is uniform, i.e., the r^{th} message on line (i, j) in the physical system is sent at time $r/\lambda'_{i,j}$, where $\lambda'_{i,j}$ is a constant independent of r. (This assumption has no justification except that it simplifies our analysis and, as demonstrated by the simulation results in the following section, it has no significant effect on the overall prediction of TBASIC's performance.)

Theorem 4: Under the above approximating assumptions,

$$D_{i,j,r} = r/\lambda_{i,j} \tag{10}$$

where $\lambda_{i,j}$ is a constant independent of r, called the *rate of message flow* along line (i, j).

$\square$

The proof of theorem 4 is given in appendix A. The proof shows how to compute the rates $\lambda_{i,j}$ of message flow on different lines in the logical system. (A message flow satisfying (10) is called a *uniform* message flow.)

In order to compute DST and SR, one would compute the values $\lambda_{i,j}$ and $M_{i,j}$ for every line (i, j), where $M_{i,j}$'s can be computed by analyzing the physical system. From this, one may compute DST and SST as follows.

$$DST = \max_{(i,j)} \{M_{i,j}/\lambda_{i,j}\} \tag{11}$$

$$SST \approx \sum_{(i,j)} (M_{i,j}/\mu_i) \tag{12}$$

SR can be computed from (1), (11), and (12).

4.2 Empirical Evaluation of the Approximate Analysis

In order to validate the above approximate analysis, we considered 10 feedforward queuing networks, and for each of them, we

1. simulated the corresponding logical system and measured its simulation time DST_m, and

2. computed the distributed simulation time DST_c of the given network using (11), and finally

3. compared DST_m with DST_c

As shown in appendix B, in each experiment the measured DST_m was within 4% from the computed DST_c. (The simulations were coded in the language MAY, which is a simulation language for distributed systems [Bagrodia 83a, Bagrodia 83b], and were run on a VAX 780. For further details concerning the simulations, we refer the reader to appendix B and [Kumar 85].)

5. Evaluation of TBASIC Using The Approximate Analysis

In this section we evaluate the performance of TBASIC in simulating several classes of feedforward queuing networks using the approximate analysis discussed in the previous section. We conclude that for many classes of these networks the speed up ratio (SR) is proportional to N, where N is the number of lps in the logical system (or equivalently the number of processors used in the distributed simulation). For the following analysis, we assume that the value of μ_i for each lp i in the logical system is the same, say μ, and that the physical system has only one source. We further assume that a delay pp can process its input messages as fast as they arrive.

The proofs of theorems 5 and 6 below, are given in appendix A.

Theorem 5: The message flow rate $\lambda_{i,j}$ on any line (i, j) in the logical system is given by

$$\lambda_{i,j} = \mu . p_{i,j}, \tag{13}$$

where $p_{i,j}$ = the *path probability* of line (i, j), i.e., the probability that a given job from the source pp would traverse this line.

$\square$

Theorem 6:

$$\mathrm{DST} = (\mu_s'/\mu).Z \tag{14}$$

$$\mathrm{SST} = (\mu_s'/\mu).Z.\phi \tag{15}$$

$$\mathrm{SR} = \phi \tag{16}$$

where Z = is the simulation time period

μ_s' = is the message flow rate from the source in the physical system, and

$$\phi = \Sigma\, p_{i,j} \text{ where the sum is over all lines (i, j)} \tag{17}$$

$\square$

It follows from theorem 6 that SR $\geq$ (N-1).p_{min}, where p_{min} = min {$p_{i,j}$, where (i, j) is a line in the logical system}. It also follows, that SR = 1 if the queuing network consists of a source followed by a sink. Otherwise, SR $\geq$ 2.

Next, we consider some specific classes of feedforward queuing networks, and determine their SR ratios in the light of equation (16).

5.1 A Tandem Network

For a tandem network of N processes, the ratio SR = (N-1). This matches our earlier result, (9).

5.2 A Parallel Network

A parallel network has K paths from a fork process to a merge process. The ith path contains L_i delay processes. (See figure 3).

Suppose one or both of the following conditions hold:

1. The branching probabilities at the fork process are all equal, or

2. L_i's are all equal.

Then it can be shown that

$$\mathrm{SR} = (3K-4)/K + N/K$$

5.3 A Serial-Parallel Network

Figure 4 shows a serial-parallel network with S stages. For any branching probabilities, we obtain:

$$SR = 3/4 .N - 1/2$$

5.4 A Full Tree Rooted at a Source

Figure 5 shows an S-stage full-tree rooted at the source. Assume that each fork process has B output branches with arbitrary branching probabilities. In this case,

$$SR = 2.\log_B[(N.B - N - B + 3)/2],$$
$$\text{for } B=2,\ SR = 2.\log_2(N+1) - 2$$
$$\text{and for } B=3,\ SR= 2.\log_3 N$$

5.5 A Non-full Tree

An example of a non-full tree is shown in figure 6. Assume that in the shown tree the branching probability at a fork for the line going to a delay is α. (The other branching probability is $1-\alpha$.) Thus,

$$SR = 2 + (1+\alpha)/(1-\alpha) .[1-\alpha^{(N/3 - 1)}]$$

Notice that,

$$\text{for } \alpha = 0,\quad SR = 3,$$
$$\text{for } \alpha = 1/2,\ SR = 2 + 3.[1 - 1/2^{(N/3 - 1)}], \text{ and}$$
$$\text{for } \alpha = 1,\quad SR = 2/3 .N$$

6. Discussion and Conclusions

Distributed simulation using the scheme TBASIC offers a substantial speed up over sequential simulation in a large number of cases. For the serial-parallel networks considered, the speed up ratio is linear with N, where N is the number of processors used in the simulation. For full trees the speed up is logarithmic. In some networks, the speed up is only a constant. However, it follows from theorem 6 that the speed up ratio is ≥ 2 in all networks (except one trivial network).

Since there are no overhead messages in TBASIC, it seems that TBASIC may offer the best performance that is possible by distributed simulation, in simulating feedforward networks. Other schemes, e.g. [Chandy 79b] and [Chandy 79a], involve overhead messages. It was noted in [Chandy 81] that some of these schemes require too many overhead messages causing their performances to degrade considerably.

Also, it seems that performance of a distributed simulation scheme would depend on

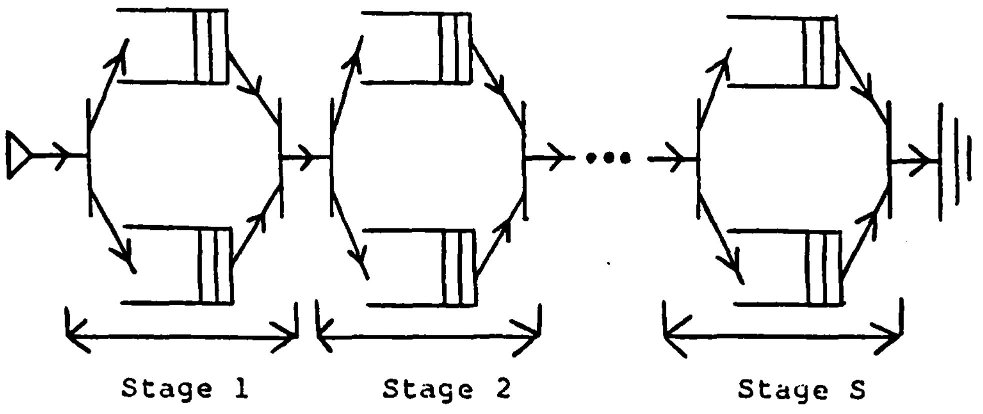


Figure 4: Serial-parallel network

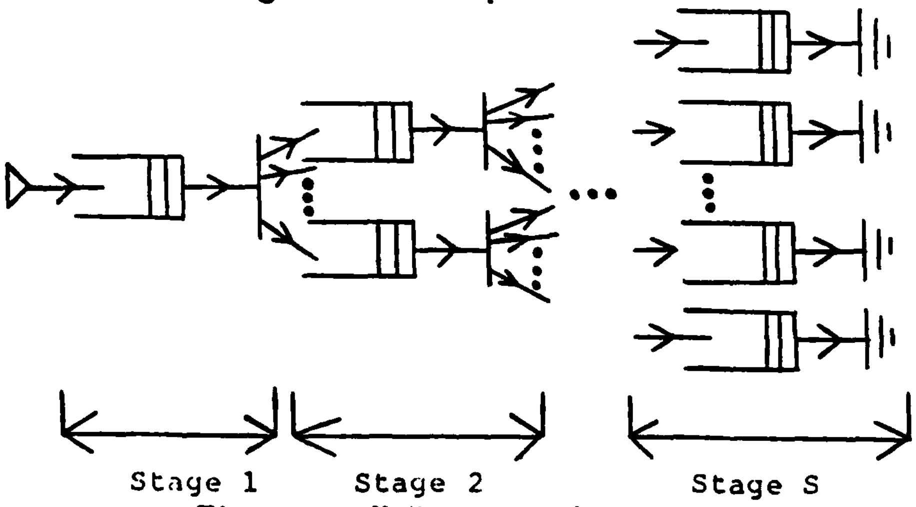


Figure 5: Full tree rooted at a source

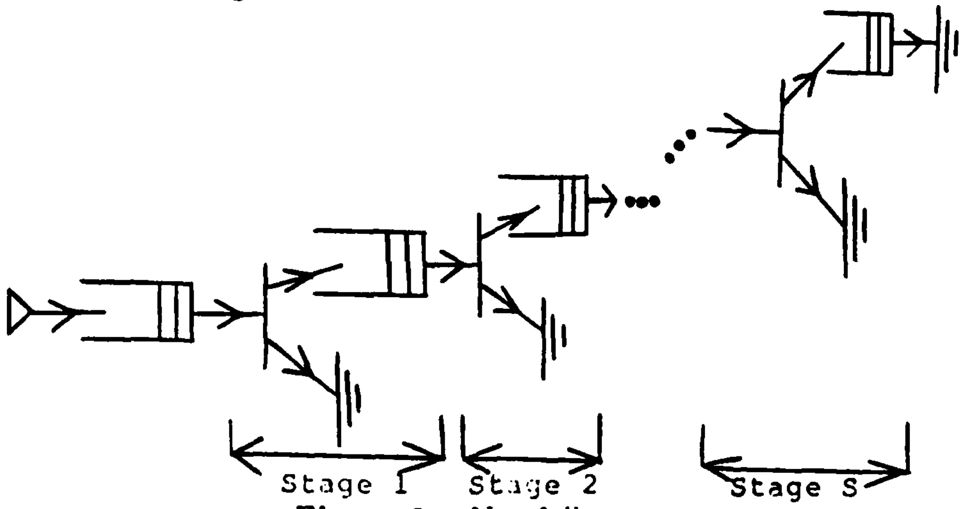


Figure 6: Non-full tree

the characteristics of physical systems being simulated. Note that TBASIC works only for feedforward physical systems. One avenue of future work would be to consider other schemes suitable for specific classes of physical systems.

Appendix A: Proofs of Theorems

Proof of theorem 2: We say that at a moment during simulation, the logical system is deadlocked if all of the following conditions hold:-

1. The t-value of at least one line is $< Z$ (the t-value of a line is the t-value of the last message sent on it),

2. there are no transient messages, and

3. every lp is either waiting for input or is terminated.

We first show that the logical system is deadlock-free. Suppose, on the contrary, the logical system is deadlocked. Consider any line (i_2, i_1) with t-value $< Z$. By the realizability property, there exists a line (i_3, i_2) with t-value $< Z$. Continuing in this manner we get an infinite sequence of lines

$(i_2, i_1), (i_3, i_2), (i_4, i_3), \ldots$

But, this contradicts the fact that the logical system has a finite number of lines and is acyclic. Therefore, we conclude that the logical system is deadlock-free.

Thus, if the t-value of a line is less than Z, then within a finite time a message would be either sent or received on some line. However, the total number of messages sent in the logical system is finite. This is based on our assumption that in the physical system the total number of messages sent on any line up to time Z is finite. On any line in the logical system there is at most one termination message; thus from the safety property of TBASIC, the total number of messages on any line in the logical system is finite. The result follows since the total number of lines in the logical system is finite.

The theorem follows, since the logical system cannot deadlock, and the total number of messages sent in the logical system is finite.

Proof of Theorem 3: We first write down the equations defining system behavior. Then we prove that (1) is the solution to this system of equations. For simplicity of discussion, let us define

$$\pi_{0,1} = 0 \text{ and}$$
$$D_{i,r} = 0 \text{ if } i = 0 \text{ or } r = 0.$$

Then, the values $D_{i,r}$ must satisfy the following system of equations:

For $i = 1, 2, ..., N\text{-}1$, and $r = 1, 2, ...$

$$D_{i,r} = \max\{D_{i,r-1}, (D_{i-1,r} + \pi_{i-1,i})\} + 1/\mu_i$$

We now prove that values of $D_{i,r}$ as given by (1) are the solution to the above system of equations. The proof is by induction on i.

Base Case:

$i = 1$.

We can establish this case by induction on r, in an obvious way.

Inductive Case:

Suppose (1) is the solution to the above system of equations for all pairs (i, r) where $i = 1, 2, ..., k$ ($k \geq 1$) and $r = 1, 2, ...$ Consider $i = k + 1$. We establish (1) for this case by induction on r.

The base case ($r = 1$) is obvious. Let us consider the inductive case ($r > 1$). Since $i > 1$ and $r > 1$, by the inductive hypotheses (on i and r), $D_{i,r-1}$ and $D_{i-1,r}$ can be obtained from (1). Thus,

$$\begin{aligned} D_{i,r-1} &= \sigma_i + \phi_i - 1/\nu_i + (r-1)/\nu_i \\ &= [\sigma_i + \phi_i + (r-1)/\nu_i] - 1/\nu_i \text{ and} \end{aligned}$$

$$\begin{aligned} D_{i-1,r} &= \sigma_{i-1} + \phi_{i-1} - 1/\nu_{i-1} + r/\nu_{i-1} \\ \text{Hence } D_{i-1,r} + \pi_{i-1,i} &= \sigma_{i-1} + \phi_i - 1/\nu_{i-1} + r/\nu_{i-1} \\ &= [\sigma_i + \phi_i + (r-1)/\nu_{i-1}] - 1/\mu_i \end{aligned}$$

Consider the following cases:

<u>**Case 1:**</u> $\mu_i < \nu_{i-1}$

$$\begin{aligned} \text{Then } \nu_i &= \mu_i < \nu_{i-1} \text{ and} \\ D_{i,r} &= D_{i,r-1} + 1/\mu_i \\ &= \sigma_i + \phi_i + (r-1)/\nu_i \end{aligned}$$

<u>**Case 2:**</u> $\mu_i \geq \nu_{i-1}$

$$\begin{aligned} \text{Then } \nu_i &= \nu_{i-1} \text{ and} \\ D_{i,r} &= D_{i-1,r} + \pi_{i-1,i} + 1/\mu_i \\ &= \sigma_i + \phi_i + (r-1)/\nu_i \end{aligned}$$

In both cases we get the same value for $D_{i,r}$ as the one given by (1). This proves the theorem.

Proof of Theorem 4: Define the *level* l(i, j) of a line (i, j) as:

l(i, j) = the maximum path length from any source lp to lp j.
(Since the logical system is acyclic, l(i, j) is well defined.)

We compute the rates of flow on the lines inductively, in increasing order of their levels. Specifically, at the kth step in the induction, we compute the rates of flow on the lines whose level = k.

Base Case (k = 1):

Any line (i, j) whose level = 1 must be an output of a source node (i.e., i is a source lp). Thus, $D_{i,j,} = r/\mu_i$.

Inductive Case (k > 1):

In this case, lp i may be a delay, fork, or merge process.

Case 1:

Pp i is a delay pp with input line (q, i).

Obviously, in this case l(q, i) = l(i, j) - 1. Therefore, by the inductive hypothesis,

$$D_{q,i,r} = r/\lambda_{q,i}$$
$$\text{Hence } A_{q,i,r} = \pi_{q,i} + r/\lambda_{q,i}$$

By induction on r, or directly from (1), it can be shown that

$$\text{Thus } D_{i,j,r} = [\pi_{q,i} + 1/\max\{\lambda_{q,i}, \ \mu_i\}] + r/\min\{\lambda_{q,i}, \ \mu_i\}$$

$$\approx r/\min\{\lambda_{q,i}, \ \mu_i\}$$

Case 2:

Pp i is a fork pp with input line (q, i) and output lines (i, j1), (i, j2), ..., (i, jn). One of these output lines is the line (i, j) with branching probability $b_{i,j}$.

Obviously, l(q, i) = l(i, j) - 1. Therefore, by the inductive hypothesis, $D_{q,i,r} = r/\lambda_{q,i}$. $D_{i,r}$, the time at which lp i sends out its rth message is $r/\min\{\lambda_{q,i}, \mu_i\}$. The proof of this is the same as in case 1.

By assumption 2,

$$D_{i,j,r} = D_{i,r}/b_{i,j}$$
$$= r/[b_{i,j} \cdot \min\{\lambda_{q,i}, \mu_i\}]$$

Case 3:

Pp i is a merge pp with input lines (l1, i), (l2, i), ..., (ln, i) and the output line (i, j).

Obviously, the level of any input line of lp i is $\leq$ (k - 1). Therefore the inductive hypothesis is applicable to the input lines.

Lp i waits till it has at least one input tuple on every line. It picks up the tuple with minimum t-value, removes it from the line buffer, and sends it out.

For simplicity of discussion, let us first assume that $1/\mu_i = 0$. We will discharge this assumption later. Consider a time interval [0, T]. For s = 1, 2, ..., n, let

r_s = the number of messages received on the line (ls, i) in this interval, and

T'_s = the t-value of the last message received on line (ls, i) in this interval.

By the inductive hypothesis,

$$r_s \approx \lambda_{ls,i} \cdot T$$

By assumption 3,

$$\begin{aligned} T'_s &\approx r_s/\lambda'_{ls,i} \\ &\approx [\lambda_{ls,i}/\lambda'_{ls,i}] \cdot T \end{aligned}$$

Suppose the minimum value of $[\lambda_{ls,i}/\lambda_{ls,i}]$, s = 1, 2, ..., n corresponds to line (lu, i). Lp i would send all the inputs received on this line; and also, from other lines, those tuples whose t-values are $\leq T'_u$. Thus the total number of messages that lp i sends out in the interval [0, T] is,

$$\approx \sum_{s=1}^{n} (r_s \cdot T'_u/T'_s)$$

$$= T'_u \cdot \sum_{s=1}^{n} (r_s/T'_s)$$

$$\approx T'_u \cdot \sum_{s=1}^{n} \lambda'_{ls,i}$$

$$=T'_u \cdot \lambda'_{i,j}$$

$$= (\lambda_{lu,i}/\lambda'_{lu,i}) \cdot T \cdot \lambda'_{i,j}$$

Thus, output flow is uniform and

$$\lambda_{i,j} = (\lambda_{lu,i}/\lambda'_{lu,i}) \cdot \lambda'_{i,j}$$

In the above discussion for case 3, we assumed that $1/\mu_i = 0$. Now let us consider the general case when $1/\mu_i$ may be non-zero. In this case, obviously

$$\lambda_{i,j} = \min \{\mu_i, [(\lambda_{lu,i}/\lambda'_{lu,i}) \cdot \lambda'_{i,j}]\}$$

This completes proof of theorem 4, and also shows how to compute the rate of flow on various lines in the logical system.

Proof of Theorem 5: For any line (i, j) in the physical system, by induction on the level of (i, j) it can be shown that,

$$\lambda'_{i,j} = \mu'_s \cdot p_{i,j}$$

where $\lambda'_{i,j}$ = the message flow rate on the line (i, j) in the physical system and

μ'_s = the message flow rate at the source pp

Using this, theorem 5 can be established, again by induction on the level of line (i, j). Note, in particular, that t-values on all input lines of a merge lp increase at the same rate (since $\lambda_{i,j}/\lambda'_{i,j}$ is independent of the line (i, j)). Thus output rate of a merge lp is the sum of its input rates.

Proof of Theorem 6: As in the proof of theorem 5, we have

$$\lambda'_{i,j} = \mu'_s \cdot p_{i,j}$$

Thus, $M_{i,j} = Z.\lambda'_{i,j}$

$$= Z.\mu'_s.p_{i,j}$$

(16) follows from the above, (14), and (12). (17) follows from the above and (13). Finally, (18) follows from (16), (17), and (1).

Appendix B: Details on Simulation of TBASIC

In this appendix we give details of our empirical study on performance of TBASIC. The physical systems simulated by TBASIC in these experiments are shown in figures 7-16. The service times at every delay pp, and the interarrival times of messages at every source pp in these experiments are chosen to be exponentially distributed, with mean values as follows:

$1/\mu_i'$ for a delay pp i, = 3000.0

$1/\mu_i'$ for a source pp i, = 4300.0

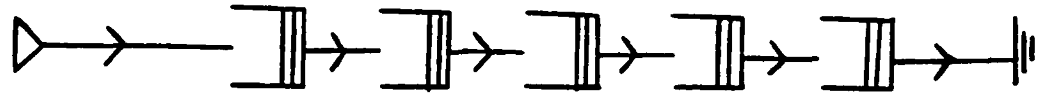
The output branches of a fork process have equal branching probabilities. The values of $1/\mu$ for various classes of pps are as follows:

delay	300
fork	90
merge	120
source	60

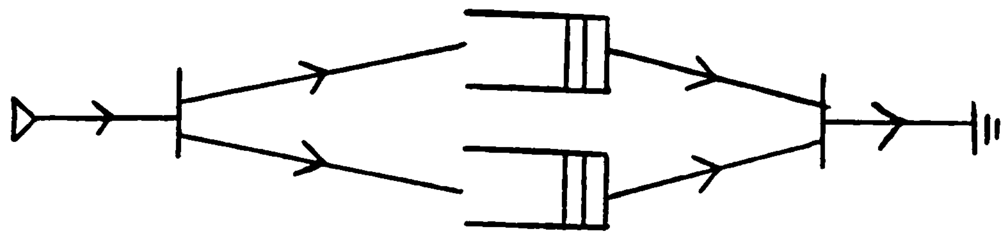
The propagation delay π on every line is assumed to be 60. The values of Z are chosen large enough to allow a large number of messages (in some experiments about 1000 and in others about 2000) to be generated by the source pp. The values of DST_m and DST_c and the *percentage errors in the values of* DST_c (deviation from DST_m)are shown in the following table. (The physical systems corresponding to these experiments are shown in figure 7.)

Exp. No.	DST_m(x 10^6)	DST_c(x 10^6)	% Error
1	0.318690	0.33	+4%
2	0.325785	0.33	+1%
3	0.326175	0.33	+1%
4	0.318180	0.33	+4%
5	0.640950	0.66	+1%
6	0.669105	0.66	-1%
7	0.681045	0.66	-3%
8	0.668805	0.66	-1%
9	0.643035	0.66	+3%
10	0.668535	0.66	-1%

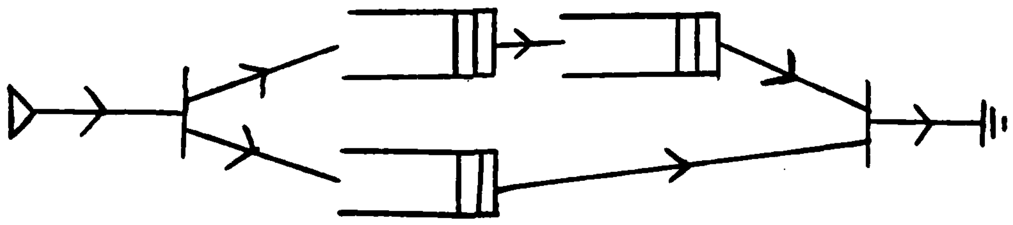
For further details on our experimental studies on TBASIC, the reader is referred to [Kumar 85].



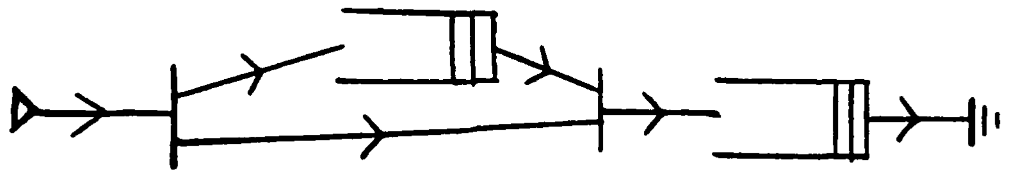
(a) Experiment 1



(b) Experiment 2

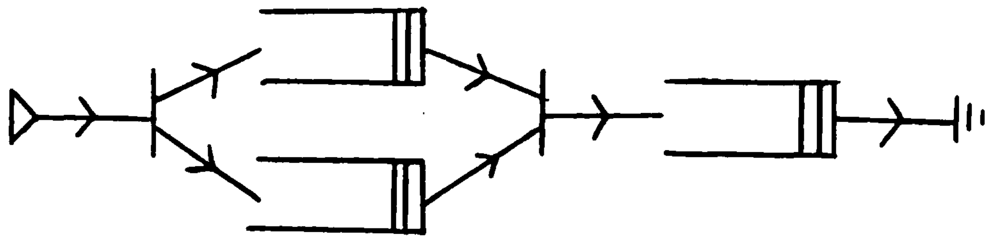


(c) Experiment 3

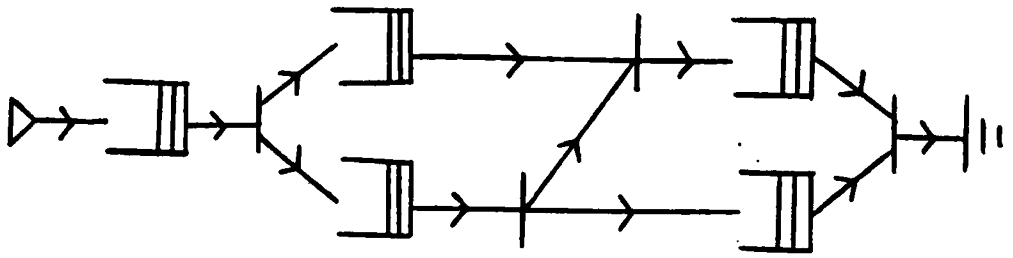


(d) Experiment 4

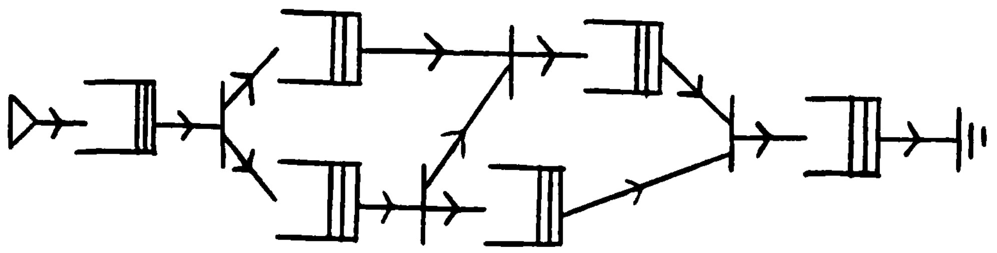
Figure 7: Physical Systems for the simulation experiments



(e) Experiment 5

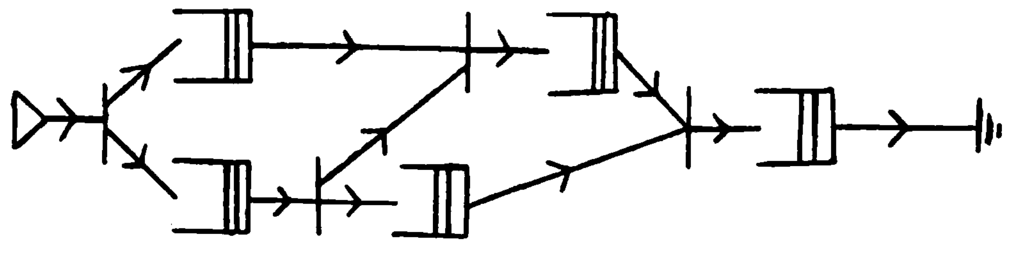


(f) Experiment 6

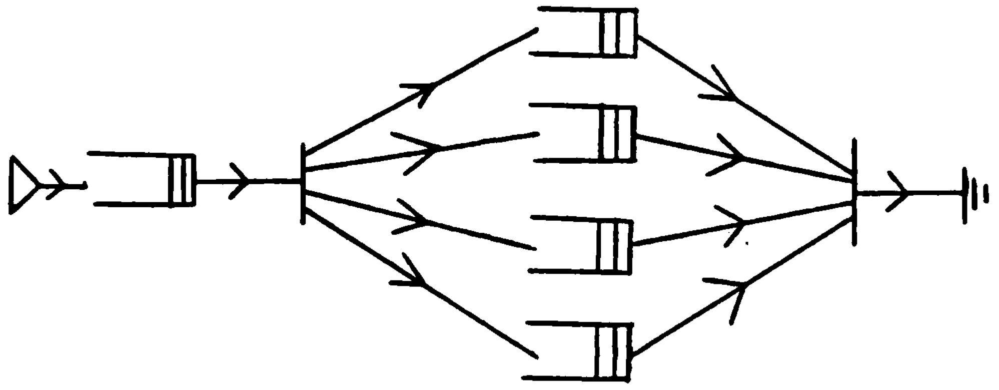


(g) Experiment 7

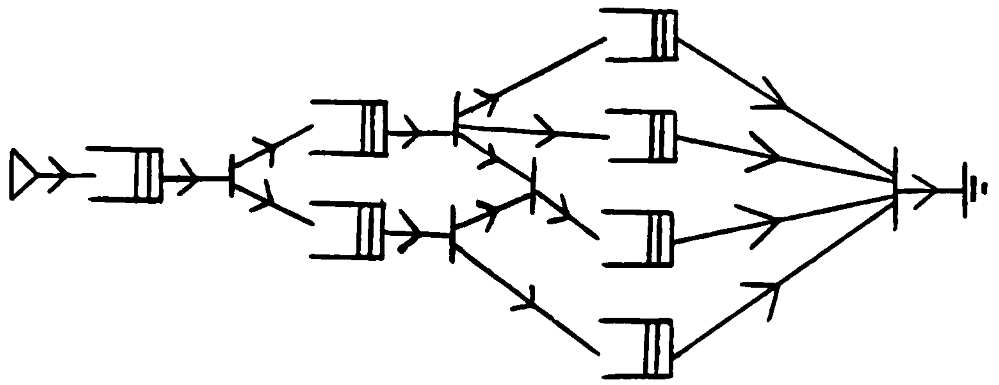
Figure 7 Continued



(h) Experiment 8



(i) Experiment 9



(j) Experiment 10

Figure 7 Continued

Acknowledgements

Professor M. Gouda made significant contributions to this paper, both in its technical content and presentation. I am thankful to Professor J. Misra for many fruitful discussions with him and for his comments on an earlier draft of this paper. I thank Rajive Bagrodia for his help in using the simulation language MAY, which considerably simplified the task of simulating TBASIC. I extend my special thanks to Ted Briggs for going over some of the proofs.

References

[Bagrodia 83a]	Bagrodia, Rajive. *May:A Process Based Simulation Language.* Masters Report, Dept. of Computer Science, University of Texas at Austin, Austin, Texas 78712, 1983.
[Bagrodia 83b]	Bagrodia, Rajive L. *MAY Reference Manual* Dept. of Computer Science, University of Texas at Austin, Austin, Texas 78712, 1983.
[Chandy 79a]	Chandy, K.M. and Misra, J. Distributed Simulation: A Case Study In Design And Verification of Distributed Programs. *IEEE Trans. on Software Engg.* SE-5(5):440-452, September, 1979.
[Chandy 79b]	Chandy, K.M., Holmes, V., and Misra, J. Distributed Simulation of Networks. *Computer Networks* 3(1):105-113, Feb., 1979.
[Chandy 81]	Chandy, K.M. and Misra, J. Asynchronous Distributed Simulation via a Sequence of Parallel Computations. *Communications of the ACM* 24(11):198-206, August, 1981.
[Jefferson 83a]	Jefferson, David R. *Virtual Time.* Technical Report TR-83-213, Computer Science Department, University Of Southern California, May, 1983.
[Jefferson 83b]	Jefferson, David R. and Sowizral, Henry A. *Fast Concurrent Simulation Using The Time Warp Mechanism, Part II: Global Control.* Technical Report, The Rand Corporation, Santa Monica, California, August, 1983.
[Kumar 85]	Kumar, Devendra. *Distributed Simulation.* PhD thesis, Dept. of Computer Science, University of Texas at Austin, Austin, Texas 78712, 1985. In Preparation.
[Misra 84]	Misra, Jayadev. *Distributed Simulation* Dept. of Computer Science, University of Texas at Austin, Austin, Texas 78712, 1984. (IEEE Tutorial on Distributed Simulation).

[Peacock 79] Peacock, J.K., Wong, J.W., and Manning, E.G.
Distributed Simulation Using A Network of Processors.
Computer Networks 3(1):44-56, Feb., 1979.

[Seethalakshmi 79]
Seethalakshmi, M.
A Study And Analysis of Performance of Distributed Simulation.
Technical Report, ut, May, 1979.